Evaluating a query framework for software evolution data

Würsch, Michael ; Giger, Emanuel ; Gall, Harald C

Abstract: With the steady advances in tooling to support software engineering, mastering all the features of modern IDEs, version control systems, and project trackers is becoming increasingly difficult. Answering even the most common developer questions can be surprisingly tedious and difficult. In this paper we present a user study with 35 subjects to evaluate our quasi-natural language interface that provides access to various facets of the evolution of a software system but requires almost zero learning effort. Our approach is tightly woven into the Eclipse IDE and allows developers to answer questions related to source code, development history, or bug and issue management. The results of our evaluation show that our query interface can outperform classical software engineering tools in terms of correctness, while yielding significant time savings to its users and greatly advancing the state of the art in terms of usability and learnability.

DOI: https://doi.org/10.1145/2522920.2522931

Evaluating a Query Framework for Software Evolution Data

MICHAEL WÜRSCH, EMANUEL GIGER, and HARALD C. GALL,
Department of Informatics, University of Zurich

With the steady advances in tooling to support software engineering, mastering all the features of modern IDEs, version control systems, and project trackers is becoming increasingly difficult. Answering even the most common developer questions can be surprisingly tedious and difficult. In this paper we present a user study with 35 subjects to evaluate our quasi-natural language interface that provides access to various facets of the evolution of a software system but requires almost zero learning effort. Our approach is tightly woven into the Eclipse IDE and allows developers to answer questions related to source code, development history, or bug and issue management. The results of our evaluation show that our query interface can outperform classical software engineering tools in terms of correctness, while yielding significant time savings to its users and greatly advancing the state of the art in terms of usability and learnability.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments, Interactive environments, Programmer workbench*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*; I.2.3 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Semantic networks*

General Terms: Human Factors, Languages

Additional Key Words and Phrases: Software Evolution, Software Maintenance, Source Code Analysis, Semantic Web, Natural Language, Conceptual Queries, Tool Support

1. INTRODUCTION

Over 90% of the costs during the evolution of software arise in its maintenance phase [Brooks 1995]. One of the cost drivers is that many project managers staff their best people in the product team, while keeping the junior developers in the maintenance team. The latter are often overwhelmed, not only by the fact that they have to understand the code and design when they were not part of the team that made the decision, but also by their need to quickly gain proficiency in using the many software engineering tools that their development teams rely on. Integrated development environments (IDEs), version control systems, and project trackers—each of these systems provides a plethora of features of their own. Orchestrating them to answer questions, such as *"Who has recently changed this code and why?"*, is even more demanding and often involves tedious manual browsing through, for example, change logs and bug descriptions. Tool support that deals with the information needs of developers is therefore well-appreciated, and the integration of different software repositories is a hot topic in research and among tool vendors.

However, existing approaches that enable the integration of different information sources often do not allow developers to formulate ad-hoc queries. Instead, they need

Author's addresses: Michael Würsch, Emanuel Giger, and Harald C. Gall, Department of Informatics, University of Zurich, Binzmühlestrasse 14, CH-8050 Zurich, Switzerland.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

to be explicitly configured to enable new queries. On the other hand, query languages, such as CodeQuest [Hajiyev et al. 2006] or JQuery [Janzen and Volder 2003], allow developers to formulate queries about software artifacts. These languages are usually based on an SQL- or Prolog-like syntax and effectively using them requires again considerable learning effort. According to Chowdhury, however, *"the most comfortable way for a user to express an information need is as a natural language statement."* [2004]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [Henninger 1994].

We have therefore devised a framework that allows software engineers to use *guided-input natural language* strongly resembling plain English to query for information about the evolution of a software system. This includes queries related to source code, development history, as well as to bug and issue management. The framework builds on Semantic Web technologies, in particular ontologies, to formalize the knowledge that describes the data from these different domains.

An early version of the framework was presented in [Würsch et al. 2010] but was limited to queries about static source code information only. Recently, we have extended our approach substantially and incorporated additional software evolution facets. In addition to source code, our framework now answers information needs related to the development history retrieved from version control systems, as well as bugs and issues reported in issue trackers. We have put further emphasis on data integration to enable queries that span multiple of these three facets. In this paper, we seek to answer the following three research questions:

— **RQ1:** How can we provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactical complexity?
— **RQ2:** When developers use such an interface to satisfy their information needs, are they able to successfully formulate and enter common developer questions, and can we observe an advancement over the state of the art in terms of time efficiency in retrieving the answers, as well as in the correctness of the answers?
— **RQ3:** Is the perceived usability higher for such an interface than for traditional means to access data about software systems, *i.e.,* those tools that are already provided by common IDEs, issue trackers, version control systems, and Web search engines?

At the heart of our paper is a user study, conducted with 35 subjects. Our study population, which provides a good approximation of junior developers, was given a set of 13 software evolution related tasks that we derived from common developer questions identified in the literature. The results of our study provide empirical evidence that subjects using our guided-input natural language interface achieve better performance in terms of correctness and time efficiency than with traditional software engineering tools. At the same time, the subjects are experiencing significantly higher system satisfaction.

The remainder of this paper is structured as follows: Section 2 gives a brief introduction to those concepts of the Semantic Web that the reader needs to understand in order to be able to follow the description of our approach. In Section 3, we present our framework to query software evolution knowledge with quasi-natural language. Its evaluation is discussed in detail in Section 4. Section 5 reviews existing work related to our approach and Section 6 concludes the paper.

2. THE SEMANTIC WEB IN A NUTSHELL

Berners-Lee et al. [2001] define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."*

Despite its origins, the Semantic Web is not limited to annotating webpages with meta-data. Virtually any piece of knowledge can be described in a computer-processable way by defining an ontology for the domain of discourse. According to Gruber [1993], an ontology formally describes the concepts (classes) found in a particular domain, as well as the relationships between these concepts, and the attributes used to describe them. For example, in the domain of software evolution, we define concepts, such as *User*, *Developer*, *Bug*, or *Java Class*; relationships, such as *reports bug*, *resolves bug*, or *affects Java Class*; and attributes, such as *email address of developer*, *resolution date of bug*, *severity of bug*, etc.

The Resource Description Framework (RDF) [Klyne and eds. 2004] is the data-model of the Semantic Web. The RDF data-model formalizes data based on *subject – predicate – object* triples, so called RDF statements. Such triples are used to make a statement about a resource of the real world. A resource can be almost anything: a project, a bug report, a person, a Web page, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [Berners-Lee et al. 1998]. In an RDF statement the subject is the URI of the thing (the resource) we intend to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model, information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [Prud'hommeaux and eds. 2008] can be used to query such RDF graphs.

RDF itself is domain-independent in that no assumptions about a particular domain of discourse are made. Specific ontologies have to be defined in an ontology definition language, such as the Web Ontology Language (OWL) [Dean and eds. 2004]. OWL enables the use of description logic (DL) expressions to describe the relationships between OWL classes. Instances of the latter are called *individuals* in OWL terminology and they belong to one or several classes. Class characteristics are specified by directed binary relations (predicates) called OWL properties. Object properties link individuals to individuals, whereas datatype properties link individuals to data values. Further constructs are provided for defining axioms about relationships between properties (*e.g.,* inverse relationships), global cardinality constraints (*e.g.,* functional properties), and logical property characteristics (*e.g.,* symmetric or transitive properties). Properties may also have a domain and range. A domain axiom asserts that the subject of a property statement (a triple) must belong to the specified class. Similarly, a range axiom restricts the values of a property to individuals of the specified class or, in case of datatype properties, asserts that the value lies within the specified data range. There is also the notion of annotation properties (henceforth called simply *annotations*), which are comparable to comments in source code in their purpose. *OWL DL allows annotations on classes, properties, individuals, and ontology headers* [Dean and eds. 2004].

In addition to the W3C recommendations, the Semantic Web community developed tools to maintain and process RDF data. Jena[1] emerged from the *HP Labs Semantic Web Program* and recently became an Apache incubator project. It is a Java framework for building applications for the Semantic Web and provides a programmatic environment for RDF and OWL. Reasoners, such as Pellet [Sirin et al. 2007], can be used in conjunction with Jena to infer logical consequences from a set of asserted facts or axioms. RDF databases, such as Apache Jena TDB, store RDF triples natively and can be queried directly with SPARQL.

[1] http://incubator.apache.org/jena/

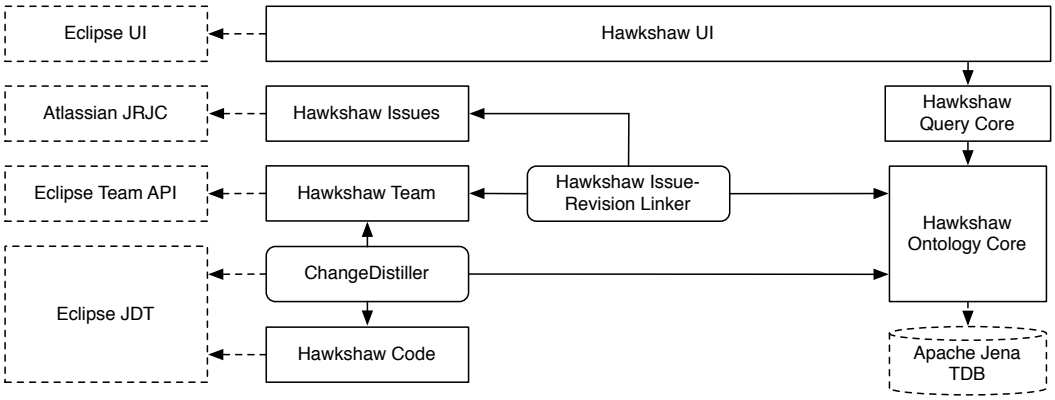

Fig. 1. Hawkshaw Component Overview

3. A QUASI-NATURAL LANGUAGE INTERFACE FOR SOFTWARE EVOLUTION DATA

In an earlier publication, we presented a framework for software engineers to answer common program comprehension questions with *guided-input natural language* queries [Würsch et al. 2010]. Our approach supported queries about source code, such as those compiled by Sillito et al. [2006]. Meanwhile, we have extended our framework to deal with questions related to various other facets of the evolution of a software system to support a wider range of common information needs of developers. This includes—besides source code—the development history, as well as the bugs and issues reported for large programs. The framework consists of a guided-input natural language interface, a set of ontologies that provide a formalization of software evolution knowledge, and a set of fact extractors to populate the ontologies with instances of real software systems. By *fact extractors* we mean parsers and algorithms that import software meta-data or *facts* from various software repositories, transform the extracted information into an ontology format, and store the results in a queryable knowledge base. Our approach is called HAWKSHAW[2] and Figure 1 gives an overview of its main components. Each box represents a component, whereas the arrows denote dependencies. Rounded corners are used for components that mainly serve as integrators for the data produced by the components they depend on. Boxes with dashed lines stand for third-party components. In the following sections, we explain each component briefly, starting with the HAWKSHAW query core that provides the algorithms that are used to guide developers in query composition.

3.1. Query Composition

The HAWKSHAW approach follows a method coined *Conceptual Authoring* or WYSIWYM (What You See Is What You Meant) by Hallett et al. [2007] and Power et al. [1998]. This means that, for composing queries, all editing operations are defined directly on an underlying logical representation, an ontology. However, the users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology.

Figure 2 shows two screenshots of our query interface. The left one shows a live example of the query dialog where a user has already started to compose a query: three words have been typed in so far, *"What Method invokes,"* and the drop-down

[2]We named our framework after Hawkshaw the Detective, a comic strip popular in the first half of the 20th century. *Hawkshaw* meant a detective in the slang of that time.

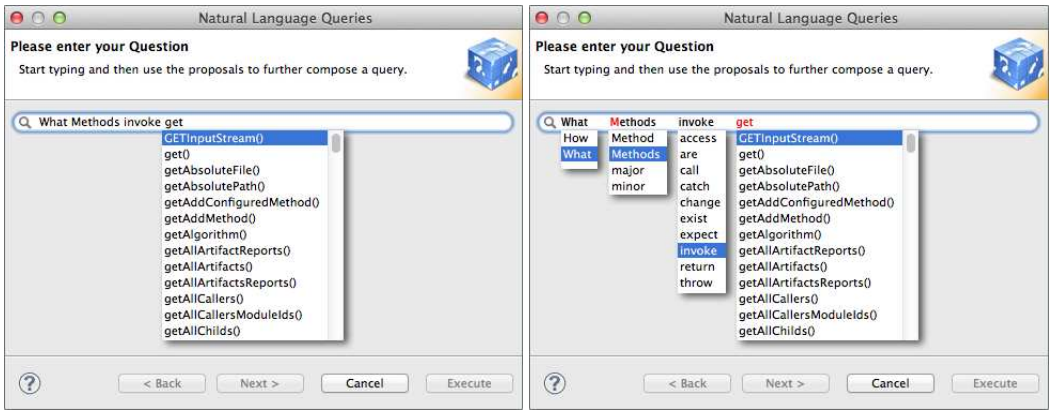

Fig. 2. The Guided-Input Natural Language Interface. The left screenshot shows a live example, whereas the right one shows how the list of proposals changed over time to reach the state shown in the left screenshot

menu presents the full list of concrete method names extracted from the source code that can be entered to complete the query. The right screenshot shows how the list of proposals changed over time before it reached its current state. The red letters mark the characters that were actually typed in by the user. The black ones were added by the auto-completion mechanism after a word was selected from the list of proposals. The list starts with the two words *"What "* and *"How."* Once *"What"* has been selected, the list will be rebuilt immediately with words that can follow the previous one. It is then again updated every time the user enters a character. In consequence, typing *"M"* will filter the list for words that start with that letter, such as *"Method"* and *"major."*

Users can type freely, as long as the entered characters match at least one of the proposed words. Therefore, our approach guides developers closely in formulating their information needs in a way such that the resulting query is processable by our query system. Our guided, quasi-natural language approach bears the following main advantages over free-form natural language queries: it is relatively light-weight in that it uses no linguistic processing at all (*i.e.,* no part-of-speech-tagging, stemming, etc.). Furthermore, developers—thanks to the proposals we show them—quickly receive feedback about the range of possible queries, and they are prevented from entering invalid questions not understandable by our query system. The immediate feedback helps to overcome the *habitability problem* formulated by Thompson et al. [2005], which occurs when there is a mismatch between the users' expectations and the capabilities of a natural language system.

Our implementation was originally based on the Ginseng tool by Bernstein et al. [2006] but we have re-developed the interface and the underlying query composition system from scratch and integrated it seamlessly into the Eclipse IDE. The query dialog shown can be brought up anytime by pressing a shortcut. It is part of the HAWKSHAW UI component and also available from the *Search* menu of Eclipse. The integration goes as far as that users can directly reference editor selections of Java entities in their questions (*i.e.,* they can enter, for example, *"Where is **this** method called?"*, after having highlighted a method in the Java editor). The re-implementation was necessary to fulfill the scalability and flexibility requirements imposed by our user study. What remains from Ginseng is that we still use a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part generated at run-time from a knowledge base. The knowledge base is formalized with an ontology described in OWL and the data is serialized by means of RDF triples. The

static part of the grammar needs to be defined manually and also contains information on how to compile the user input into the SPARQL query language.

The static grammar is part of the HAWKSHAW query core and consists of rules that basically act as a template for possible questions. The rules contain placeholders which are replaced on-the-fly during query composition by natural language labels and phrases extracted from the knowledge base. The labels and phrases constitute the dynamic part of the grammar, which is part of the HAWKSHAW ontology core. Table I lists nine out of over 80 rules taken from our grammar definition files. In practice, the rules follow a notation similar to the *Extended Backus-Naur Form* but, for the sake of presentation, we introduce a simplified notation and omit the formal part needed for translation into SPARQL: Arrows denote that the right word or phrase follows after the left one, expressions in square brackets mark non-mandatory parts of a rule. Phrases in quotation marks represent statically encoded natural language parts. Each rule is terminated by a punctuation mark, in particular a period or question mark. Italic words are placeholders for dynamic rules generated from the ontology (*i.e.,* non-terminal symbols). The symbol $noun_c$ stands for nouns that are extracted from annotations of the OWL classes in our ontology (see Section 3.2 for more details on the annotations). Occurrences of $noun_i$ refer to labels of OWL individuals, *i.e.,* the instances of the OWL classes. The $verb$ and $adjective$ symbols are replaced by annotations of OWL object- and boolean datatype properties, respectively. Another case is $noun_{attr}$, which stands for annotations of non-boolean datatype properties.

Curly brackets can be used to mark the subject of a sentence, but are optional as long as a rule contains only one verb. The brackets can therefore be omitted in all but one of the example rules (Rule 5). The additional information is used together with the domain and range restrictions defined for the OWL object properties in our ontology to filter inappropriate words from the list of proposals. In other words, the verb of the sentence has to be an object property that fits the subject. That means that the object property needs to have the class in its domain that has been selected during query composition as the subject of the sentence. Object properties not fulfilling this constraint will not be presented to the user. Similarly, the object of the sentence must be an individual of a class in the ontology. The individual's class has to comply to the range of the object property, or it will not be shown either. For rules where the subject is not explicitly marked, we assume the last entered noun is the subject of the sentence.

Consider the following example, where a developer wants to find all methods that invoke another method with the identifier `bar()` and access a given field named `foo` (Rule 4 in Table I). When the query dialog is brought up, a proposal list with words that are allowed to begin a question will pop up: *"Are"*, *"List"*, and *"What."* Then either a word can be selected from the list by clicking on it, or users can start to type in the letters. While typing, no longer relevant words are automatically filtered from the list. For example, if the word *"What"* is entered, Rules 2, 6, and 9 will no longer be relevant. The query system will then retrieve the next set of proposals according to the remaining six rules from the underlying knowledge base, in this particular example by retrieving all annotations of boolean datatype properties (*adjective*), as well as those of OWL classes ($noun_c$), since the *adjective* symbol is marked optional. Once the word *"method"* is selected, the query system will rebuild the proposals with *verbs* that are allowed to follow the previous noun, *i.e.,* with the annotations of those OWL object properties that have the OWL class `Method` in their domain restriction. One such object property is `invokesMethod`, which we have annotated in the ontology with verbs, such as *"invokes"* and *"calls."* Once selected, the range restriction of `invokesMethod` limits the next values for the $noun_i$ symbol to labels of those individuals that are instances of an appropriate OWL class. If the developer continues to compose the question like this, eventually the possibility will arise to either complete the sentence with a question

Table I. Static Grammar Rule Examples

#	Rule	Example Question
	Open Questions	
1	"What" | "How many" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow$ "are there" | "exist" $\rightarrow$?	How many unresolved bugs are there?
2	"List" | "Give me" $\rightarrow$ "all" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow$.	List all protected methods.
3	"What" | "How many" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow verb \rightarrow$ ["the" $\rightarrow$ $noun_c \rightarrow$] $noun_i \rightarrow$?	What abstract classes implement the interface IFoo?
4	"What" | "How many" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow verb$ [$\rightarrow$ "the" $\rightarrow$ $noun_c \rightarrow$] $noun_i \rightarrow$ [and $\rightarrow$ $noun_i \rightarrow$] ?	What critical bugs are blocked by #123 and #124?
5	"What" | "How many" $\rightarrow$ [$adjective \rightarrow$] $\{noun_c\} \rightarrow verb$ [$\rightarrow$ "the" $\rightarrow$ $noun_c \rightarrow$] $noun_i \rightarrow$ [and $\rightarrow verb \rightarrow noun_i \rightarrow$] ?	What methods invoke bar() and access foo?
	Closed questions	
6	"Are there any" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow$ "that" $\rightarrow verb \rightarrow noun_i \rightarrow$?	Are there any public methods that call bar()?
	Superlatives	
7	"What" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow verb \rightarrow$ "the" $\rightarrow$ "most" | "least" $\rightarrow noun_c \rightarrow$?	What class is affected by the most bugs?
	Comparisons	
8	"What" $\rightarrow$ [$adjective \rightarrow$] $noun_c \rightarrow verb \rightarrow$ "more" | "less" $\rightarrow$ "than" $\rightarrow number \rightarrow noun_c \rightarrow$?	What method is called by more than 20 methods?
	Aggregations	
9	"List" $\rightarrow$ "the average" $\rightarrow noun_{attr} \rightarrow$ "of all" $\rightarrow noun_c \rightarrow$.	List the average size of all methods.

mark, or to continue by adding the conjunction *"and."* In the first case, the question will be compiled into a SPARQL query. The formal query is then automatically executed against the knowledge base and the results are displayed in a view similar to the one used by Eclipse for its Java Search results. In the second case, the query system will first retrieve annotations of the individuals represented by the $noun_i$ symbol that fit the last *verb* (Rule 4), as well as those object properties represented by *verb* that fit the $noun_c$ which was flagged as subject of the sentence (Rule 5). Then the query composition process continues until again the rule allows the question to terminate by the question mark.

In our previous work [Würsch et al. 2010], we have shown that our framework already enables a wide range of queries comparable to that of other state-of-the-art research tools, such as the approach by de Alwis and Murphy [2008]. At the same time, HAWKSHAW exhibits more flexibility because it is not limited to a small set of concrete, hard-coded queries. For this paper, we have extended its querying capabilities even further, with additional knowledge extracted from version control systems or bug and issue trackers. The knowledge is used to generate additional dynamic rules, which—in combination with the static ones—allow for queries, such as *"What critical bugs affect this method?"* or *"Which developers changed this class?"* In the next section, we explain how we formalized the knowledge so that it can be queried with our approach.

3.2. SEON—Software Evolution Ontologies

Our approach makes heavy use of Semantic Web technologies. In the Semantic Web, knowledge is represented with ontologies, which in turn are described in terms of triples of subject, predicate, and object. This structure strongly resembles how humans talk about things and it can be easily transformed into natural language sentences.

Specializations and generalizations allow to query knowledge on multiple levels of abstraction; for example, one can ask for *"person"* and also retrieve instances of developers and testers. Properties in OWL represent a binary relation that can be restricted by specifying domain and range. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, the explicit semantics can be exploited to filter the verbs that can follow a given subject, or the objects that can follow a given verb. All these features render ontologies a valuable knowledge representation format to be used in our approach.

The acronym SEON stands for *Software Evolution ONtologies*. It represents our attempt to formally describe knowledge from the domain of software evolution analysis and mining software repositories [Würsch et al. 2012]. SEON covers a multitude of concepts but for the work presented in this paper only those related to source code, development history, issues and bugs, as well as fine-grained changes are of relevance.[3] In the following, we give a brief overview on these ontologies. For further details, we refer to the original paper about SEON.

Code Ontology. The source code ontology provides a formal meta-model for those source code entities and dependencies, which are common to many object-oriented programming languages. The formalization was done through the definition of OWL classes, object- and datatype properties. For example, we define OWL classes to represent packages, types, fields, methods and their parameters, etc. Object properties describe their relationships: classes are declared in a file and they declare members—methods and fields. The classes can inherit from other classes, methods invoke other methods, and so on.

Issue Ontology Modern project trackers often provide bug and issue management features. The data generated by those trackers is described by our issue ontology. The key concept is that of an issue, which can have several specializations, such as bugs, feature requests, improvements, etc. Both the key concept and its specializations are represented by OWL classes. Issues have a description, usually written by their reporters. Eventually, they are assigned to developers for fixing them. Often, issues are triaged and classified by priority and severity. In consequence, we defined object properties, such as `fixesIssue`, `hasAssignee`, `hasPriority`, and `hasSeverity`, as well as the datatype properties `hasIssueNumber`, `hasDescription`, etc.

History Ontology. Data stored in version control repositories is represented by the ontology about the development history of software systems (*i.e.,* revisions or versions, releases, etc.). The central concept is that of a version of a file, which we have covered with the definition of an OWL class. Versions of files are committed by one specific developer and they have a commit message. A particular version of a file can be part of a release.

Change Ontology. Most version control systems are not aware of the exact syntax of the changes they maintain. Instead, they work on file-level and track only textual changes, *i.e.,* updates, additions, and deletions of lines. Our fine-grained change ontology describes changes made to source code down to the program statement level. For example, it allows us to describe the addition of a method invocation statement to a constructor of a class between two versions of a software system.

Natural language annotations provide human-readable labels for all classes and properties in SEON. They therefore bridge the gap between the machine-processable ontologies and the end-users of our quasi-natural language interface. For individuals, we use RDF Schema labels (rdfs:label) that are generated by our fact extractors. For example, the rdfs:labels of methods are simply their Java identifiers followed by parentheses, *e.g., "println()."* The OWL class representing the concept related to Java

[3]The full OWL definitions of SEON can be browsed online at: `http://se-on.org/`

methods has a custom OWL annotation property with the value *"method,"* whereas the object property `invokesMethod` describing a caller-callee relationship has the three annotations *"invokes," "calls,"* and *"uses."* These words are used synonymously during query composition. HAWKSHAW extracts at runtime the natural-language annotations to guide developers in formulating questions, such as *"What method invokes...?"*, *"What method calls...?"*, and *"What method uses...?"*. By proposing a variety of synonyms, we take into account that developers use different nuances in terminology. With a reasonable selection of synonyms, HAWKSHAW can offer an experience that comes very close to free-form natural language input.

3.3. Fact Extraction

Next, we briefly describe how we populate our ontologies with instance data from real software systems under development. This happens with the aid of different fact extractors that first obtain data from various software repositories and then analyze different facets of the evolution of a software system. The fact extractors are implemented as a set of plug-ins for the Eclipse IDE and the extracted instance data is stored in a file-based Apache Jena TDB triple store. The integration of the different facets into one queryable software evolution knowledge base is described in Section 3.4.

Source Code Analysis. We use the Eclipse Java Development Tools (JDT) for the extraction of the static source code facts. In particular, we have implemented a custom project builder that maintains an up-to-date ontology model of the source code of a Java project stored in the local workspace of an Eclipse installation. The performance of the builder is comparable to that of the JDT builder during compilation of Java code. Parsing roughly 100k lines of code, including the creation and storage of the associated ontology model in TDB, takes less than one minute on a typical laptop computer.

Historical Analysis. To import and analyze the history of a project under version control, we interface the Eclipse Team API and retrieve, for each file, the full commit history. In consequence, with the HAWKSHAW tool, one is able to query the development history of any project no matter what version control system it uses—as long as there is a Team API compliant Eclipse plug-in available for that system. Analysis performance, however, strongly depends on the type of repository used. For example, the import of six years of development history with roughly six thousand different versions took about 20 minutes from a remote SVN repository. The analysis of the same number of versions, but from a GIT mirror of the same project, completed in under two minutes. The reason for the notable difference is that the SVN plug-in sends out many HTTP-requests, so that network performance becomes a limiting factor. For each GIT clone, in contrast, the whole change log is already available locally in compressed form.

Fine-Grained Change Extraction. During historical analysis, we also run our fine-grained change extraction algorithm on each pair of consecutive versions [Fluri et al. 2007]. CHANGEDISTILLER uses the Eclipse JDT to build an abstract syntax tree for each version and creates an edit script to transform the source code of the older version into that of the newer one. The tree edit operations in the script are then classified into change types, such as *statement insert*, *method renaming*, and so on. The performance overhead of the change extraction is negligible on modern personal computers.

Issue Extraction. We have implemented a fact extractor for the Atlassian JIRA project tracker that is based on the JIRA REST Java Client (JRJC). The extraction performance depends on the network connection. For example, we imported over one thousand issues in under ten minutes from the Apache Foundation tracker.

3.4. Integration and Reasoning

Issue trackers and version control systems are information silos, with little to no integration between them and no possibility for performing cross-domain queries [Tap-

```
CONSTRUCT
   { ?v seon:fixesIssue ?i . }
WHERE
   { ?v seon:hasCommitMessage ?message .
     ?i seon:hasKey ?key .
     FILTER regex(?message, xpath:concat(?key, "[^0-9]"), "i") }
```

Fig. 3. SPARQL Construct Query for linking Revisions to Issues

polet 2008]. To overcome this limitation of current software repositories, we perform additional integration steps to link issues to changes, as well as changes to code.

To find links to the issue database, we scan the commit messages of each version for references to bug and issue numbers. The scanning and linking can be done with a single, concise SPARQL construct query—at least for projects that define a rigid change process, with developers consistently referencing issues in each commit. The corresponding SPARQL query is shown in Figure 3. When it is executed, the graph pattern consisting of the two triple patterns and a `FILTER`-expression in the `WHERE`-clause is matched against the triples of the RDF graph and returns the bindings for the variables in the `CONSTRUCT`-clause. SPARQL variables are indicated by the prefix "?" and, within a graph pattern, a variable must have the same value no matter where it is used. In the given query, the first pattern will match any triples where the predicate is the property `seon:hasCommitMessage`. Since the domain and range definitions of this property restrict the possible values of the subject and object, only revisions will be bound to `?v` and their commit messages to `?message`. Similarly, the second pattern will match against any statement with `seon:hasKey` as predicate and, consequently, the bindings for `?i` will contain issues and those for `?key` the corresponding issue keys generated by the issue tracker. The filter expression uses a regex function to narrow down the set of matching statements to those where the commit message of the revision contains the key of the issue. For each pair of revisions and issues, the `CONSTRUCT`-clause will result in a new triple with the revision as subject, the property `seon:fixesIssue` as predicate, and the issue as object. We then add all resulting triples to our triple store.

Linking between versions and source code changes is done during fact extraction with CHANGEDISTILLER, where we explicitly state, for each extracted change, which source code entity was modified in what version. Now that issues are linked to versions, and the latter to code, we use the Pellet reasoner [Sirin et al. 2007] to bridge the gap between source code changes and the issues that most likely caused them. For that, we defined rules, such as the following: When method m changes in version v, and version v is linked to issue i, then i affects m. The reasoner will apply that rule automatically to our ontology model and add the resulting triples to our triple store. Since we also use the reasoner to infer inverse properties (*e.g.,* `affectsIssue` is an inverse of `isAffectedByIssue`), we can then propose a multitude of domain-spanning questions to developers, for example *"What issues affected this method?"* or *"What classes were affected by issue #123?"*.

In summary, our approach combines industrial-strength technologies with ideas and tools from the Semantic Web to enable queries about software evolution artifacts in a way that comes natural to developers: using (quasi) natural language strongly resembling plain English. We use OWL to describe different software evolution artifacts and the relationships between them. A reasoner helps to make implicit knowledge explicit and therefore queryable. The resulting knowledge base then serves as input for our HAWKSHAW query interface. With the proof-of-concept implementation of HAWKSHAW,[4]

[4]Hawkshaw is available for download at: http://se-on.org/hawkshaw/

Table II. Hypotheses

	Null Hypotheses		Alternative Hypotheses
$H1_0$	The distribution of the *Total Score for all Tasks* is the same across the experimental and control group.	*H1*	The distribution of the *Total Score for all Tasks* is different across the experimental and control group.
$H2_0$	The distribution of the *Total Number of Seconds spent for solving all Tasks* is the same across the experimental and control group.	*H2*	The distribution of the *Total Number of Seconds spent for solving all Tasks* is different across the experimental and control group.
$H3_0$	The distribution of the *System Usability Scores* is the same across the experimental and control group.	*H3*	The distribution of the *System Usability Scores* is different across the experimental and control group.

we can answer our first research question, RQ1: yes, with the components described in this section, it is possible to provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactic complexity. In the next section, we present an extensive user study to evaluate the remaining two research questions.

4. USER STUDY

Our vision is to provide a convenient and intuitive interface that allows software engineers to access various kinds of knowledge related to the evolution of their software systems. To evaluate whether HAWKSHAW meets this claim, we designed and carried out a user study with 35 participants. In particular, we sought to answer the two remaining research questions with our evaluation:

— **RQ2:** When developers use such a quasi-natural language interface (*i.e.,* HAWKSHAW) to satisfy their information needs, are they able to successfully formulate and enter common developer questions, and can we observe an advancement over the state of the art in terms of time efficiency in retrieving the answers, as well as in the correctness of the answers?
— **RQ3:** Is the perceived usability higher for such an interface than for traditional means to access data about software systems, *i.e.,* those tools that are already provided by common IDEs, issue trackers, version control systems, and Web search engines?

We laid out the user study as a *Between Subjects Design* where the subjects were randomly assigned to either an experimental or a control group [Wohlin et al. 2000]. From the 35 subjects that participated in our study, 18 were assigned to the experimental and 17 to the control group. The experimental group was provided with HAWKSHAW. For the control group, we prepared a reasonable set of common developer tools. These tools served as a baseline to compare our approach against. The selection of the baseline is discussed thoroughly in Section 4.4.

We then assigned the same set of 13 software evolution tasks to both groups and defined three hypotheses based on RQ2 and RQ3 to statistically validate the outcome of our study. The software evolution tasks are introduced in Section 4.1 and our hypotheses are listed in Table II.

4.1. Choosing the Tasks

Finding a representative set of tasks is fundamental for the validity and generalizability of a user study. We therefore surveyed the literature for previous experiments and existing catalogues of common developer questions in the context of program comprehension or software maintenance and evolution. Our aim was to find relevant questions related to source code, development history, and issues, which would allow us to directly compare HAWKSHAW with tools that are widely used in industry. We further looked for questions spanning multiple facets to evaluate our integrated approach.

We compiled our set of 13 tasks from the work of LaToza et al. [2006], Sillito *et al.* [2006; 2008], Ko et al. [2007], de Alwis and Murphy [2008], Fritz and Murphy [2010], and Hattori et al. [2011]. These tasks are listed in Table III. We anonymized developer names, issue numbers, Java identifiers, etc., for publication but the full questionnaire can be obtained from the corresponding author of this paper. With the rationale listed in the table, we describe the importance of each task in our own words.

We intended to stay as close as possible to the original text of the questions from the literature but found it necessary to adapt them with light modifications: the questions were made more specific (where necessary) in order to reduce the range of possible interpretations. For example, the high-level question *"What have my coworkers been doing?"* [Ko et al. 2007] was reformulated into *"What feature requests were implemented by Developer$_2$?"*. The original question could also be interpreted in numerous other ways, *e.g., "What classes or files were committed by Developer$_X$?"*, so we removed the potential source of confusion for the study subjects to facilitate scoring of correct answers.

Because of HAWKSHAW's conception as a quasi-natural language interface, developers can enter many other questions found in the literature without further transformations. While this is an important feature of our approach, we still tried to re-formulate the questions to neutral sentences so that we did not treat our approach preferentially. In a few cases, we added twists to evaluate specific aspects. *"What calls this method?"* [de Alwis and Murphy 2008], which can be entered directly in HAWKSHAW, became Task 3: *"All methods that invoke both, Method$_1$ and Method$_2$."*

In general, we paid close attention not to penalize the baseline tools in comparison to our approach. For example in Task 3, where the intersection of the callers of two different methods A and B is requested, we selected methods with a low number of callers to facilitate the composition of the partial results for the control group. One of the methods had 15 different callers, but the other one had only two—coming up with the common callers, hence, was trivial. Similarly for Task 6: *"The last five files changed by Developer$_1$?"*, where we looked for a developer that had recently committed five files, so that the participants of the control group did not have to browse through many revisions. In fact, the result was to be found already among the five top-most rows of the *History View* of Eclipse.

We decided to exclude questions concerning the fine-grained change history provided by CHANGEDISTILLER from the user study. The decision was made after we had completed a pre-study, which showed us that such tasks—especially in comparison with our approach—are poorly supported by Eclipse and therefore hardly solvable within the given time. Questions, such as "What developers changed Method$_X$?", can be answered with HAWKSHAW right-away but often involve laborious differencing with the *Compare*-feature in Eclipse.

In summary, we are convinced that we selected tasks that do not unduly favor our HAWKSHAW tool and that these tasks relate to valid, common information needs of software developers. To further support this claim, we additionally asked the participants of our user study to rate each task with respect to its degree of realism, *i.e.,* whether they would be likely to solve a task similar to the one at hand in practice. These ratings are presented in Section 4.9, however they have to be considered with care because our subjects were mostly students with limited industrial experience.

4.2. Evaluating Usability

We used standardized satisfaction measures to investigate how user-friendly our study subjects perceived the HAWKSHAW approach. The same measures were applied to the baseline tools for comparison. After a thorough evaluation of usability-related measures, we decided for the *System Usability Scale (SUS)* by Brooke [1996], a popular questionnaire for end-of-test subjective assessments of usability. The SUS a de-facto

Table III. Task Description and Rationale

ID	Tasks
	Code Domain
T1	**Description.** All the subclasses of $Class_1$? [Sillito et al. 2008]
	Rationale. When a base class is modified, its subclasses will be affected. Finding those classes quickly is the first step in assessing the change impact. Also helpful for program comprehension, *e.g.,* when searching for implementations of an abstract class.
T2	**Description.** All methods with $Class_2$ as argument (parameter)? [Sillito et al. 2008]
	Rationale. Identifying methods that can operate on an instance of a given class can reveal useful API that supports the task at hand.
T3	**Description.** All methods that invoke both, $Method_1$ and $Method_2$? [Sillito et al. 2008]
	Rationale. Finding pieces of code where a method is referenced is crucial for program understanding and also for finding proper API usage examples. Often, it is also helpful to combine the result from two distinct searches, *e.g.,* when checking for violations of common idioms.
	History Domain
T4	**Description.** The developers who have changed $Class_3$ in the past? [Hattori et al. 2011]
	Rationale. Team activity awareness is helpful for coordination and for finding experts of a particular part of a software system.
T5	**Description.** The file that has changed most often in the past? [Fritz and Murphy 2010]
	Rationale. An excessive amount of changes can indicate a design problem and reveal candidates for refactoring in order to improve the separation of concerns.
T6	**Description.** The last five files changed by $Developer_1$? [Hattori et al. 2011]
	Rationale. When one has to replace a previous project team member, it is important to quickly get an overview on the member's previous work. One way to achieve this, is by looking at the code the team member was working on.
	Issue Domain
T7	**Description.** What feature requests were implemented by $Developer_2$? [Ko et al. 2007]
	Rationale. Similar to Task 6, the goal is to become familiar with someone else's work. This time from a different angle, *i.e.,* by looking at the kind of features the developer was responsible for.
T8	**Description.** The issues $Developer_3$ and $Developer_4$ commented on? [Fritz and Murphy 2010]
	Rationale. When relying on third-party libraries, developers often comment on bug reports in those libraries that affect their work. Retrieving these bugs later is useful for various reasons, such as team awareness, time tracking, and checking if workarounds are no longer necessary.
T9	**Description.** The issues blocked by $Issue_1$? [common feature of issue trackers]
	Rationale. As soon as an issue is resolved, other issues previously blocked by the current one can move into the center of attention.
	Cross-Domain
T10	**Description.** The classes affected by $Issue_2$? [LaToza et al. 2006]
	Rationale. Quickly assessing the impact of an issue is useful for effort measurement.
T11	**Description.** The issues that affected $Class_4$? [LaToza et al. 2006]
	Rationale. Understanding change history if a piece of code is the first step in understanding the design decisions behind its implementation. Previous issues affecting the code explain some of the reasons for change.
T12	**Description.** The most error-prone class? [Kim et al. 2007]
	Rationale. Kim *et al.* have reported *that faults do not occur in isolation, but rather in bursts of several related faults.* In consequence, past bugs are a good predictor for future bugs. Finding classes affected by many bugs therefore can help allocating resources for testing efficiently.
T13	**Description.** The issues that affected $Class_5$ and $Class_6$? [LaToza et al. 2006]
	Rationale. Finding issues affecting different classes can reveal the reasons behind logical coupling.

industry standard, used in hundreds of publications, and its robustness and reliability have been confirmed empirically by, amongst others, Bangor et al. [2008].

An advantage of the SUS over alternatives, such as the *Post-Study System Usability Questionnaire (PSSUQ)* [Lewis 1992], is its conciseness; it can be filled out quickly by the subjects, lowering the risk of incomplete or non-serious responses. The SUS consists of the following ten items, each with five response options that range from "Strongly Disagree" to "Strongly Agree":

(1) I think that I would like to use this system frequently.
(2) I found the system unnecessarily complex.
(3) I thought the system was easy to use.
(4) I think that I would need the support of a technical person to be able to use this system.
(5) I found the various functions in this system were well integrated.
(6) I thought there was too much inconsistency in this system.
(7) I would imagine that most people would learn to use this system very quickly.
(8) I found the system very cumbersome to use.
(9) I felt very confident using the system.
(10) I needed to learn a lot of things before I could get going with this system.

We asked our subjects, after they had completed all the 13 tasks, to record their immediate response to each item (as recommended by the author of the SUS), rather than thinking about items for a long time. The experimental group had to answer with respect to their experience with HAWKSHAW, whereas the control group had to reflect on the features of the baseline toolset that they actually used for solving the tasks. We then aggregated the responses into a single number for each subject, representing a composite measure of the overall usability of HAWKSHAW and the baseline, respectively. The exact scoring is described in Section 4.7.

In addition to the measurements provided by the post-test questionnaire SUS, we measured user satisfaction immediately after the completion of each task. Among the numerous questionnaires available to gather post-task responses, we selected the *Single Ease Question (SEQ)*. According to Sauro and Dumas [2009], the SEQ exhibits the important psychometric properties of being reliable, sensitive, and valid—while also being short, easy to respond, and easy to score. It consists of the single question *"Overall, this task was?"* in combination with a seven-point Likert-scale that ranges from *"Very Difficult"* to *"Very Easy"*. Sauro claims that:

> *"The beauty of the SEQ is that users build their expectations into their response. So while adding or removing a step from a task scenario would affect times, users adjust their expectations based on the number of steps and respond to the task difficulty accordingly."* [Sauro 2010]

This property is important, because it allows us to differentiate between the cases where a task was actually difficult to solve with the available tools, and those when obtaining the correct solution was simply laborious, but less mentally challenging.

4.3. Research Population

Our research population included 35 subjects, of whom 25 were advanced undergraduate students and eight graduate students. The remaining two participants were post-doctoral researchers. We recruited the subjects among the approx. 80 participants of two different courses: a fourth-semester lab course on software engineering and an advanced course on software evolution and maintenance. Since an internship in industry is part of the curriculum, the students of the advanced course already had industry-level software development experience. The experimenters were involved in neither of the two courses

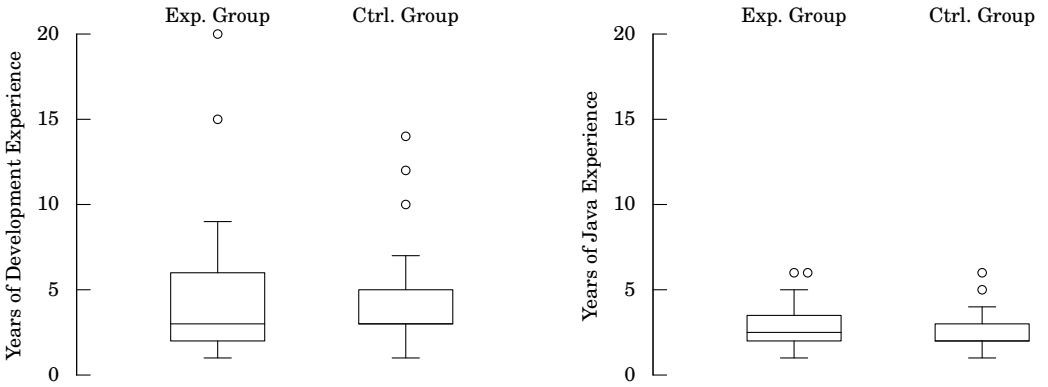

Fig. 4. Number of Years of General Development Experience (left) and Java Development Experience (right) of the Experimental Group (18 Subjects) and Control Group (17 Subjects)

and the participation in our study took place on a completely voluntary basis for our students. However, we rewarded them with a small monetary compensation for their time. We clearly communicated that we were evaluating exclusively our approach—not the participants—and that we respected the anonymity of the subjects at all times.

The average subject's age was 25.2 years, ranging from a minimum of 20 to a maximum of 39 years. In Figure 4, an overview of the number of years of development experience is given for the subjects of both groups. We report both development experience in general (*i.e.,* no matter what programming language and IDE) and development experience with Java in particular. Overall, the average development experience of the participants was five years with a median of three years, ranging from the minimum of one to a maximum of 20 years. Their particular development experience with Java and the Eclipse IDE was between one and six years, with 2.8 years on average and a median of two years.

We further asked the participants to do a quick self-assessment of their skills in English,[5] coding in general, Java in particular, JIRA, and SVN. An overview on the self-assessment of both groups can be found in Figure 5.

In summary, it shows that our research population is a good approximation to junior developers, which we expect to be the user group that benefits most from our HAWKSHAW approach.

4.4. Finding a representative Baseline Toolset for the Control Group

The tasks of our user study involved questions about source code, the development history, and issues. While our HAWKSHAW approach integrates this wide array of information in one queryable ontology model, most modern IDEs provide only a subset of the features out of the box that are necessary to access the same information directly from within the IDE. We therefore prepared a small set of commonly used developer tools as a baseline for comparison with our approach.

The baseline tools were comprised of all the features of the Eclipse Classic package (v3.7.2), the Subversive SVN plug-in for Eclipse, and a Web browser for conducting Web searches and accessing the Web front-end of the Atlassian JIRA project tracker. Eclipse provided source code browsing and search features and was already well-known to most participants of our study. The majority of our participants were already familiar

[5]The tools, as well as the questionnaire, were in English, whereas the majority of our students are native (Swiss-)German speakers.

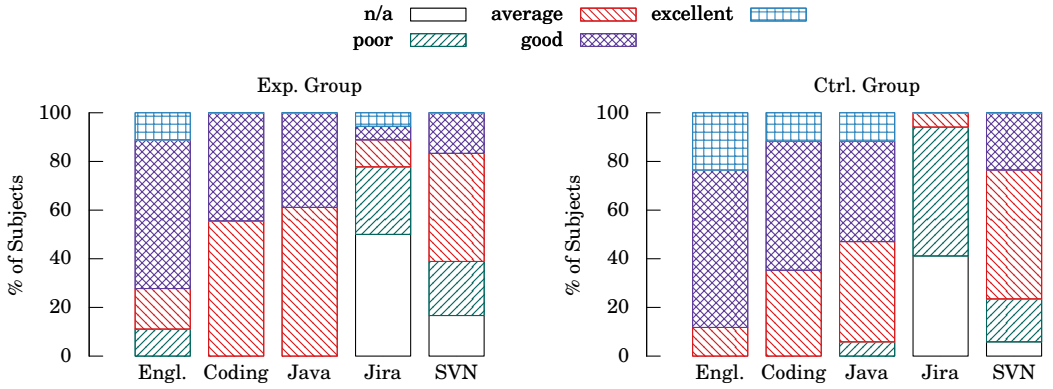

Fig. 5. Skill Self-Assessment of the Experimental Group and the Control Group

with SVN and the Subversive Eclipse plug-in, since they had actively used it in one of their lab courses for a few months in the context of a team development project. The Subversive plug-in contributes the *History View* to Eclipse that allows browsing the change log of a project or of single files.

We have chosen to study an open source project, which uses JIRA for its issue management (see Section 4.5 for details on the selected project). Besides the popularity of JIRA, we found the feature set provided by its Web front-end to be well-suited for direct comparison with our HAWKSHAW approach. Most importantly, JIRA provides tight integration with SVN: for each issue, the related SVN commits with the involved files are listed in the Web interface. The SVN integration compensates for the lack of traceability between issues and changes in standard Eclipse installations.

All these tools provide a strong baseline with a variety of features that left the control subjects well-equipped to solve the tasks listed in Section 4.1. We also claim that the baseline is representative for the tools that many developers employ nowadays in practice.

4.5. Choosing a Study Object

The tasks presented in Section 4.1 were carried out by the participants on a real software system: we selected the Apache Ivy project,[6] consisting of 103,647 source lines of code (SLOC) in 919 Java classes at the time of the user study. It is a popular open-source dependency manager and has been used as a case study in an increasing number of publications recently, for example, in [Posnett et al. 2011; Basit et al. 2011]. Ivy has been under development for about six years under the patronage of the Apache Foundation, where a team of seven developers has contributed to it. We have analyzed and imported with HAWKSHAW more than six thousand revisions of Java files from the SVN repository of Ivy and more than one thousand issues from its JIRA tracker.

Our selection was based on the consideration that the subjects should answer questions about a real-world, industry-scale project to increase the external validity of our study. However, only such systems were considered by us that were written in Java and neither exceeded a reasonable size, nor belonged to a very complex application domain. These restrictions were imposed by the short amount of time the participants had to familiarize themselves with the system, as well as by the average skill and experience level of our participants.

[6] http://ant.apache.org/ivy/

We further looked for projects that use SVN for version control and JIRA for issue tracking, and that implement a rigid change process. For such projects, our fact extractors can reliably link JIRA issues to SVN revisions. Both criteria are met by Ivy: its developers use SVN and each commit message contains a reference to the corresponding change requests (*i.e.,* the issue numbers).

4.6. Conducting the User Study

We carried out the user study of HAWKSHAW during three sessions. Each session lasted for approx. 45 to 80 minutes. The study took place in the computer labs of the Department of Informatics at the University of Zurich, where we had installed HAWKSHAW and the baseline tools in advance.

Each session started off with a short introduction, where we outlined the goals of the study. We quickly went through each page of the questionnaire to make sure that the members of the experimental and control group understood what we were asking of them. The subjects were explicitly told what tools they could use and which they were not allowed to rely upon. The complete instructions were replicated on the front-page of the questionnaire, which existed in two variants: one for the experimental- and one for the control group. The tasks for both groups were identical but we gave the control group some hints on which of the baseline tools could help in solving each subset of tasks. For example, for the tasks involving questions about issues, we mentioned that the JIRA tracker of the Apache Ivy project could yield interesting insights and listed its Web address. Or, as another example, the *History View* of Eclipse was suggested for solving the tasks related to the development history.

We prepared a short tutorial for each group in the form of a PDF document which we provided to the participants at the beginning of each session. It contained annotated screenshots of the tools, as well as some brief explanations of the tools' features needed for succeeding in the study. The tutorial for the experimental group explained the HAWKSHAW query interface. Additionally, it contained a few general tips for effective querying, *e.g.,* that the subjects should only formulate questions with subject-predicate-object structure and in present tense. We gave them only two positive and one negative examples for queries with HAWKSHAW—all three of them unrelated to the tasks in the study. In the control group's tutorial for using the baseline tools, we explained the source code search features of Eclipse. This included a brief user guide on the *Type Hierarchy View*, the *Call Hierarchy View*, the *Find-References* command, as well as on the *Show-History* command and the associated *History View*. We further demonstrated the use of the JIRA Web front-end by labeling all important widgets on screenshots and explaining how the simple and the advanced search works. The detailed tutorial in conjunction with their previous Eclipse and SVN experience left the members of the control group in a highly-competitive starting position compared to their counterparts in the experimental group, which could only rely on our tutorial but never had used HAWKSHAW before.

We did not restrict the time for reading the tutorial and it was up to the subjects to start working on the tasks when they felt ready for it. However, once they began solving the first task, we imposed strict time limits on them: for each task, the subjects were given five minutes. Once this amount of time passed, the participants had to write down their answer and then proceed to the next task.

4.7. Data Collection and Pre-Processing

Throughout the course of the study, we collected various data, which we then pre-processed and analyzed to obtain the empirical results described in Section 4.8. The collection and pre-processing steps are outlined below.

Personal Data. Prior to the study, we asked the participants to share some personal information with us, such as age, gender, current occupation, and their English skills. We also enquired about their expertise with software development in general, as well as about their Java and Eclipse skill in particular. Finally, they were also asked about their level of familiarity with SVN and JIRA.

Correctness Data. To map each subject's solutions to scores comparable with those of others, we used the following simple scoring scheme: each of the tasks, when solved correctly, was rewarded with one point. So a maximum score of 13 points could be achieved. In case of incomplete solutions, each correct item was worth one point divided by the number of total correct items. For example, the correct solution to Task 7 consists of two Java classes. In case that a subject wrote down only one answer out of two, we scored the solution with 0.5 points. We decided not to penalize incorrect answers; in the previous example, if the subject wrote down a third unrelated class, the total score for that task was not diminished. The model solution (or oracle) was defined by the authors with aid of the baseline tools. Then it was carefully validated with HAWKSHAW to ensure that no false positives or true negatives remained. In the remainder of the paper, we will use the term *correctness* whenever we talk about these scores; if, for example, we say that group x achieved an average correctness of 6.5, we mean that the participants of the group received on average 6.5 out of 13 possible points.

Timing Data. We asked the participants to solve the tasks as quickly as possible and in the given order. To time the subjects accurately, we contributed an *Experiment Timer View* to Eclipse. The view showed the current task and a progress bar with the remaining time for solving that task. The subjects had to start the timer themselves, once they had read the tutorial and provided their personal information. In case the timer expired, a popup and an audible notification urged the participants to write down their (partial) answer and proceed to the next task. When the participants finished a task early, they could click a button to proceed. In any case, the application displayed a reminder to write down the elapsed time on the questionnaire before restarting the timer for the next task.

Usability Data. To measure usability of HAWKSHAW in comparison to the baseline tools, we incorporated the SUS questionnaire (*cf.* Section 4.2). The SUS yields a single number between zero and 100, representing a composite measure of the overall usability of the system being studied. The score is based on the individual answers for each of the ten items of the SUS. We then applied the original SUS scoring scheme:

> *To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4. For items 1, 3, 5, 7, and 9 the score contribution is the scale position minus 1. For items 2, 4, 6, 8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by 2.5 to obtain the overall value of SU.* [Brooke 1996]

Besides the classical SUS score, which represents a global measure of system satisfaction, recent research has discovered empirical evidence for two sub-scales of usability and learnability [Lewis and Sauro 2009]. In particular, the fourth and tenth items provide the learnability dimension and the other eight items provide the usability dimension. For both, the contributions per item are calculated based on the scoring scheme above. However, the sum of the items related to learnability and the sum of those related to usability is then multiplied by 12.5 and 3.125, respectively.

Qualitative Feedback. In addition to the quantitative data gathered, we asked the subjects for their opinion on our quasi-natural language approach, as well as on the user study itself. In particular, we asked for any comments and/or suggestions that could improve the user study. The members of the experimental group additionally could comment on what they liked the most/least of HAWKSHAW. They were also given

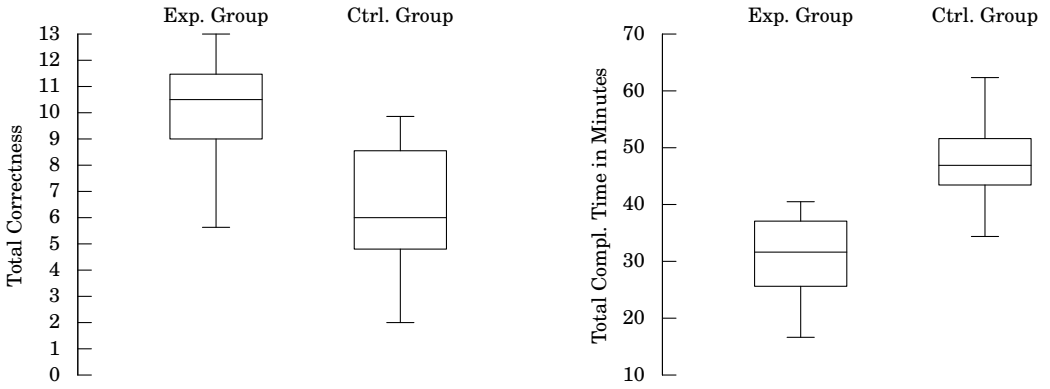

Fig. 6. Total Correctness (left) and Total Completion Time (right) per Group

the opportunity to list any questions that they wanted to enter with our approach but did not succeed in doing so. The control group was asked for features of Eclipse and JIRA that were most helpful to them during the study, and whether they were missing any functionality or found something particularly difficult to use.

4.8. Empirical Results: Overview

We first present the statistical analysis of the aggregated results of our user study. In Section 4.9 we then describe the data we obtained for each of the individual tasks. The results of the study are summarized in Section 4.12 and we discuss potential threats to validity that may have arisen from our study design in Section 4.11. The interpretation and discussion of the empirical results is given in Section 4.12.

To support our choice of an appropriate statistical test for our hypotheses, we analyzed the distributions of the overall results on correctness, completion time, and usability scores—as well as the data for each individual task. Although the *Shapiro-Wilk Test of Normality* hinted at normally distributed data, a visual analysis of the corresponding histograms and the Q-Q plots raised reasonable doubts on normal distribution—especially when looking at how the individual results for each task were distributed. We performed global tests to accept or reject the hypotheses given in Table II. In addition to the global tests, we performed post-hoc tests for the individual tasks to break down the overall results into detailed results for every task. We therefore decided to use the non-parametric Independent Samples Mann-Whitney U (MWU) test with a significance level $\alpha = 0.01$. The post-hoc tests are presented in Section 4.9. Their purpose is to identify how each task contributes to the overall results.

Overall Correctness. The subjects in the experimental group achieved on average a 59.13% higher correctness than those in the control group: the mean total correctness for the experimental group is 10.20 out of 13 possible points with a standard deviation (std dev.) of 2.00 and a median of 10.50. For the control group, we observed a mean of 6.41 with a std dev. of 2.27 and a median of 6.00. The left box plots in Figure 6 further illustrate that the 25^{th} percentile of the experimental group lies above the 75^{th} percentile of the control group, which means that 75% of the subjects in the experimental group reached a higher total correctness than 75% of their counterparts in the control group.

We tested the two distributions for equality. The MWU test showed that their difference is significant at the 99% confidence level (p-value=2.66E-5). As a consequence, we reject $H1_0$ (*cf.* Table II) and accept the alternative hypothesis that the distribution of the total number of correct answers is different across the two groups.

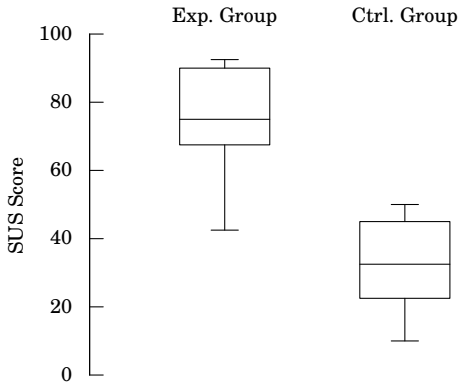

Fig. 7. System Usability Score per Group

Overall Completion Time. With our approach, participants were 35.41% faster than the baseline. This is equivalent to average time savings of 16 minutes and 54 seconds for all the 13 tasks. The experimental group needed, on average, 30 min 50 sec to solve all tasks—with a std dev. of 6 min 61 sec and a median of 31 min 37 sec. The mean for the control group is 47 min 44 sec, the std dev. is 7 min 4 sec, and the median is 46 min 54 sec.

In Figure 6, we depict similar results with respect to completion time, as we had previously made for correctness: The 75^{th} percentile of the box plot for the experimental group is below the 25^{th} of the box plot for the control group. In other words, three quarters of all the subjects with HAWKSHAW could solve the tasks faster than three quarters of the subjects with the baseline tools.

The MWU test rejects the null-hypothesis $H2_0$ at the 99% confidence level with a p-value of 6.13E-8. The acceptance of the alternative hypothesis confirms that the distribution of the total completion times among the two groups is different. With the acceptance of $H1$ and $H2$, we can now answer RQ2: developers can indeed successfully formulate and enter common developer questions with our quasi-natural language interface. Overall, HAWKSHAW provides a clear advancement over the baseline tools with respect to both correctness and time efficiency.

Overall Usability. The mean of the SUS score for HAWKSHAW is 42.02 points higher than the mean of the control group's score, that is 76.11 versus 33.09 out of a theoretical maximum score of 100. The std dev. for the first group is 14.46, and for the second one 14.81. This translates to high sub-scores for HAWKSHAW in usability (mean=74.31, std dev.=15.18, and median=73.43) and learnability (mean=83.33, std dev.=16,61, and median=87.5), whereas the baseline tools were rated significantly lower in both usability (mean=31.8, std dev.=14,91, and median=34.38) and learnability (mean=38.24, std dev.=21.86, and median=37.5).

The box plots in Figure 7 show that the 25^{th} percentile of the experimental group is again above the 75^{th} percentile of the control group, denoting that HAWKSHAW was well-accepted by the majority of its users, whereas the subjects of the control group were, overall, less satisfied with the baseline tools.

Sauro [2011] provides guidance in interpreting SUS scores based on a comprehensive study. In analyzing data from over 5000 users across 500 different evaluations, the author determined an average score of 68 for the systems tested and also calculated the percentile ranks for different ranges of scores. The SUS score obtained for HAWKSHAW lies clearly above the average. In fact, it converts to a percentile rank of 73%, which

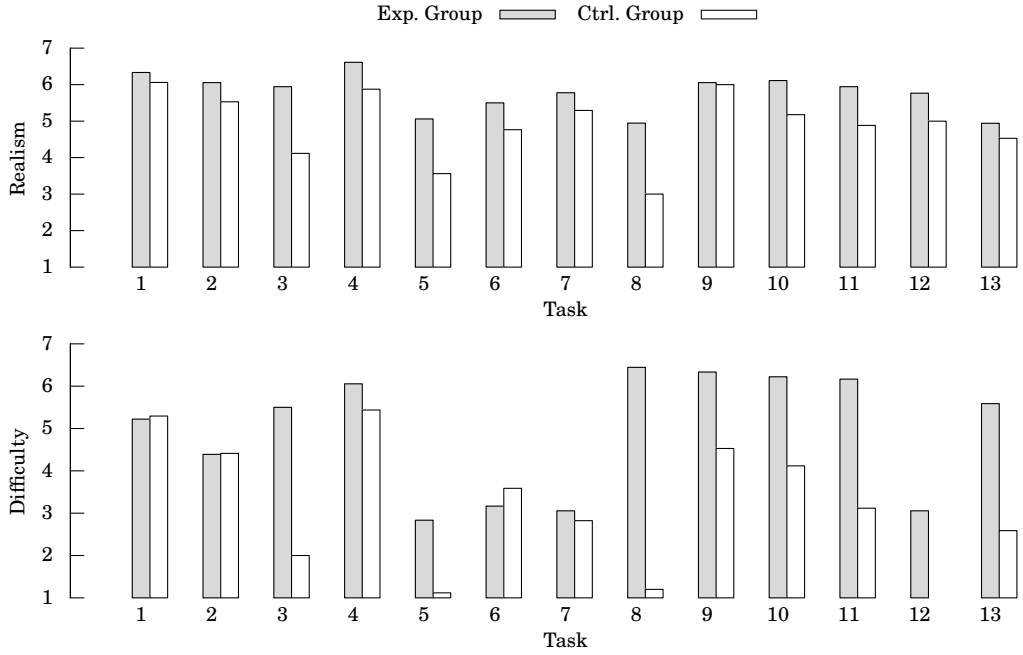

Fig. 8. Realism (1 - very unrealistic, 7 - very realistic) and Difficulty (1 - very difficult, 7 - very easy) per Group and Task

means that HAWKSHAW has a higher perceived usability than 73% of the other systems tested.

We also assessed the reliability of the SUS in our user study with Cronbach's Alpha—a measure widely used in social sciences to calculate the internal consistency of psychometric tests. High alpha values basically indicate that *"answers to a reliable survey will differ because respondents have different opinions, not because the survey is confusing or has multiple interpretations."* [Norusis 2010]. In our case, we calculated a value of 0.937, which is commonly considered as an indicator for excellent internal consistency. This high alpha value is in line with those values that have been previously reported in the literature for the SUS [Bangor et al. 2008; Lewis and Sauro 2009].

The MWU test rejects $H3_0$ at the 99% confidence level (p-value=2.95E-8). We therefore accept $H3$: the SUS scores are different for the experimental and control groups. As a consequence, we can now answer our last research question, RQ3: Yes, the perceived usability of HAWKSHAW is significantly higher than that of traditional tools.

4.9. Detailed Task Analysis and Interpretation

The overall results presented in Section 4.8 show a clear advantage of our approach over the baseline tools. To analyze where the advancements come from, we performed post-hoc tests for correctness and completion time for each task individually. Because of the post-study character of the SUS, we could not break down the overall system satisfaction into individual scores for each task. However, to still obtain individual results, we additionally surveyed the subjects after each task with the post-task SEQ questionnaire. Furthermore, we assessed the practical relevance of each task by asking the subjects whether they found the task realistic or not. An overview on the usability scores per task and group can be found in the lower chart of Figure 8, whereas the upper one shows how the two groups rated the practical relevance of each task.

Table IV. Individual Results per Task

		Correctness (Points)				*Completion Time (Min:Sec)*			
ID	Group[a]	Mean	Mean$_\triangle$[b]	Median	StdDev	Mean	Mean$_\triangle$	Median	StdDev
T1	exp.	0.89	+0.01	1.00	0.32	03:10	+00:27	03:08	01:22
	ctrl.	0.88		1.00	0.28	02:43		02:45	01:11
T2	exp.	0.69	-0.03	1.00	0.45	03:18	+00:14	03:04	01:07
	ctrl.	0.72		1.00	0.40	03:05		03:07	01:07
T3	exp.	0.83	+0.42	1.00	0.38	02:18	-02:06**	01:56	01:06
	ctrl.	0.41		0.00	0.51	04:24		05:00	00:53
T4	exp.	0.94	+0.01	1.00	0.24	01:48	-00:30	01:46	00:38
	ctrl.	0.91		1.00	0.26	02:18		02:10	01:09
T5	exp.	0.17	+0.17	0.00	0.38	03:27	-01:33**	03:01	01:13
	ctrl.	0.00		0.00	0.00	05:00		05:00	00:00
T6	exp.	0.51	-0.18	0.60	0.49	03:23	+00:22	03:25	01:35
	ctrl.	0.69		1.00	0.46	03.01		03:09	01:29
T7	exp.	0.75	+0.25	1.00	0.39	03:53	-00:36	04:13	01:11
	ctrl.	0.50		0.55	0.44	04:29		04:35	00:31
T8	exp.	1.00	+1.00**	1.00	0.00	01:18	-03:19**	01:09	00:33
	ctrl.	0.00		0.00	0.00	04:38		05:00	00:55
T9	exp.	0.94	+0.09	1.00	0.24	00:59	-01:32**	00:46	00:36
	ctrl.	0.85		1.00	0.29	02:31		02:18	01:31
T10	exp.	0.94	+0.31	1.00	0.24	01:08	-02:16**	00:59	00:42
	ctrl.	0.63		1.00	0.48	03:24		02:45	01:26
T11	exp.	0.94	+0.53**	1.00	0.24	01:23	-01:54**	00:57	01:02
	ctrl.	0.41		0.33	0.43	03:24		03:10	01:20
T12	exp.	0.61	+0.61**	1.00	0.50	03:18	-01:26**	03:17	01:37
	ctrl.	0.00		0.00	0.00	04:44		05:00	01:06
T13	exp.	0.97	+0.58**	1.00	0.10	01:26	-02:44**	01:23	00:26
	ctrl.	0.39		0.43	0.41	04:10		04.48	01:18

** high significance (global $\alpha = 0.01$, Bonferroni-Holm method applied)
[a] experimental group = exp., control group = ctrl.
[b] $Mean_\triangle = Mean_{exp} - Mean_{ctrl}$

Both the post-hoc tests for correctness and completion time as well as the tests to compare the perceived difficulty and realism levels were performed with the MWU test, unless stated otherwise. The visual analysis of the distribution of the data for the individual tasks suggested non-normality and therefore called for a non-parametric test. We applied the Bonferroni-Holm method for the post-hoc tests, which is a sequentially rejective version of the simple Bonferroni correction for multiple comparisons. It strongly controls the family-wise error rate at level alpha and *"ought to replace the classical Bonferroni test at all instants where the latter usually is applied."* [Holm 1979]. The global confidence level remained at 99%.

Table IV lists the mean, mean difference, median, and std dev. values for the correctness and completion time in seconds per group and task. The differences that are statistically highly significant according to the MWU test are marked with two stars (**). Figure 9 depicts the results; it shows the mean correctness and completion time per group and task, as well as the corresponding std dev. values. What follows is a brief summary of the statistics obtained for each task, as well as our interpretation of these results.

Task 1: Learning about class hierarchies—All the subclasses of *Class*$_1$?

Results. The difference between the experimental and control groups is statistically insignificant in both correctness and completion time. The subjects of both groups agreed that the the first task was easy to solve with the available tools and that the task was realistic—in both cases with insignificant differences in their ratings.

Interpretation. Displaying and navigating a class hierarchy is well-supported in Eclipse through its *Type Hierarchy View*. The usage of the view was explicitly explained

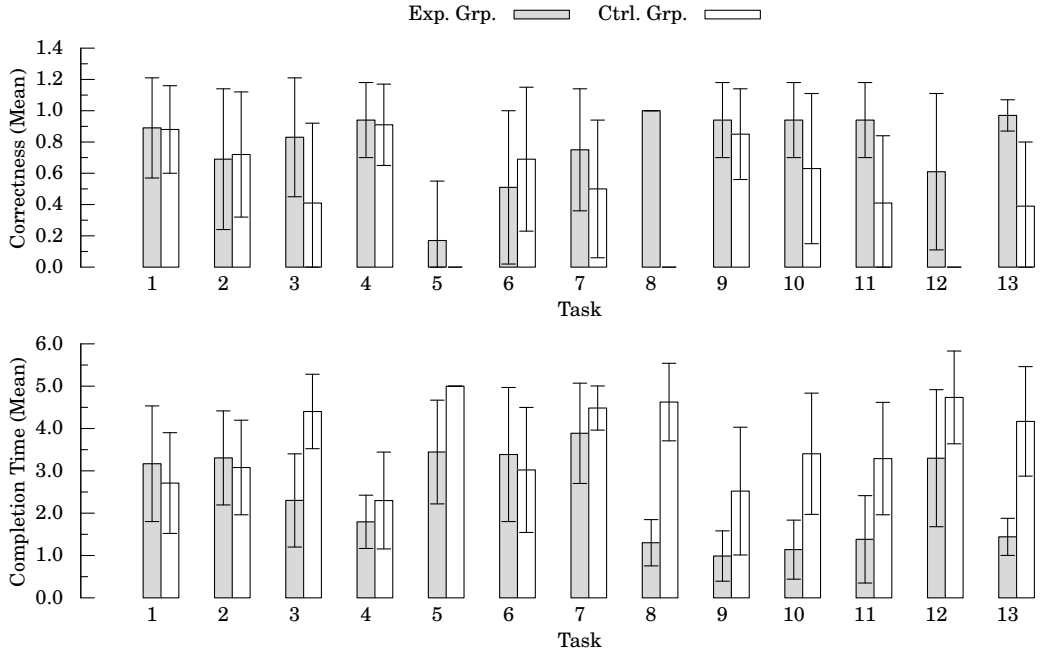

Fig. 9. Mean Correctness (top) and Completion Time (bottom) per Group and Task. The Error Bars show the Standard Deviation.

to the control group in their tutorial. In light of this, it is remarkable how well HAWKSHAW competed with Eclipse, especially since most subjects were already familiar with Eclipse, whereas none of them had used our tool so far. In addition, we intentionally stated the task description in a way that forced the subjects to reformulate the given sentence for HAWKSHAW. The goal was to test whether they could make use of the query composition guidance provided by our approach. In particular, the noun *"subclass(es)"* had been omitted[7] from our natural-language annotations and, instead, the question had to be entered as *"What classes extend...?"* or *"What classes inherit from...?"*. Unsurprisingly, a common complaint by the subjects of the experimental group therefore was that our approach did not directly allow for questions, such as *"What are the subclasses of...?"*. Despite this twist, 16 out of 18 subjects of the experimental group succeeded in quickly submitting a correct query. For unknown reasons, the other two persons wrote down only the (correct) package names while unfortunately omitting the class names. In such cases of doubt, we decided against HAWKSHAW—and in favor of the baseline tools—and counted the answers as incorrect.

Task 2: Finding methods that operate on an instance of a given class—All methods with $Class_2$ as argument (parameter)?

Results. We could not observe any notable difference between the two groups, neither in terms of correctness and completion time, nor concerning their difficulty ratings. On average, the control group found it less plausible that they would have to solve such a task in practice. However, the difference to the experimental group is statistically insignificant.

[7]Meanwhile we added the concepts *Superclass* and *Subclass* to SEON. For any triple *"a extends b"*, a reasoner will now automatically classify *a* as *Subclass* and *b* as *Superclass*.

Interpretation. Finding methods that use a given type as a parameter can be achieved in Eclipse in two ways: through the *References*-feature in the context-menu, or by using the *Java Search* dialog. Both features were illustrated with screenshots in the control group's tutorial. Nonetheless, the simplicity of our query interface was able to compensate for the head start of the baseline tools. Particularly for this task, we received ambiguous qualitative feedback on one feature of HAWKSHAW: our tool by design does not allow for fully qualified type or method names in questions. Instead, users have to reference Java entities simply by their identifier. While some subjects told us that they appreciated this feature because it resembles closely how they reference types and methods when they talk to colleagues, others were confused that they did not find the full qualifiers among the proposals provided during query composition. The resolution was straight-forward: we now support both short identifiers and fully qualified names.

Task 3: Finding the common callers of two methods—All methods that invoke both, *Method*$_1$ and *Method*$_2$?

Results. The mean correctness achieved by the experimental group is roughly twice as high as that of the control group (0.83 vs 0.41). However, when performing the MWU test in conjunction with the Bonferroni-Holm correction, the null-hypothesis is retained with a p-value of 0.032 at a corrected α-level of 0.001. Hence, the difference is deemed insignificant. We therefore binned the answers into fully correct (correctness=1.0) and incorrect ones (correctness $<$ 1.0) to be able to perform Pearson's Chi^2 test on the null-hypothesis that the correctness is independent of the subject's group. With a p-value of 0.001, the null-hypothesis is rejected at a confidence level of 99%. For completion time, the results are more conclusive: the experimental group was able to solve Task 3 significantly faster than the control group (MWU test, corrected α=0.001, p-value $<$ 9.69E-6). These results translate to those on realism and difficulty: the experimental group found the task very realistic and very easy to solve. The control group, in contrast, rated the degree of realism on average with about four (most subjects were undecided), whereas they perceived the level of difficulty as very high.

Interpretation. Finding code that invokes a given method is widely recognized in the literature, *e.g.,* by Sillito et al. [2008], as a common task among developers and corresponding search features are incorporated in almost any modern IDE. However, we increased the difficulty level of the task by adding a conjunction: we asked for the common callers of two methods, instead of simply the callers of a single method. Hence, we tested the control group's ability to compose the information fragments of two Java searches, while the experimental group could simply chain the parts in one question. The difference in completion time and perceived difficulty is nonetheless surprising because, of the two methods given in the task, the second one had only two callers in the whole system—applying the right search strategy, the correct answer could be obtained in matter of seconds. However, we observed that multiple subjects of the control group only searched for the references of the first method and then browsed the code of the 15 results for invocations of the second one. This poor search strategy lead to the higher completion times observed for the control group. The potential difference in correctness could be a result of failure to check all the callers of the first method within the allotted time. However, given the inconclusive results of the statistical tests, we refrain from drawing a final conclusion on the correctness in Task 3.

Although the composition of information fragments is an important field of research in software engineering [Fritz and Murphy 2010], our task is artificially more challenging. It is however notable that we explicitly faced a similar question during a maintenance task, when we were checking for violations of a locking-related idiom to resolve a concurrency issue in HAWKSHAW.

Task 4: Learning about code ownership—The developers who have changed *Class*$_3$ in the past?

Results. Both groups performed equally well with respect to correctness and completion time. They attributed the task a very high degree of realism and a very low level of difficulty.

Interpretation. The *History View* of Eclipse provided the control group with a quick and easy to read, table-based view on the change log of individual files. Also the experimental group faced no difficulties in reformulating the task description into a query understandable by HAWKSHAW. Notable was that two subjects mentioned that they looked at the proposals presented by our approach and were unsure about the difference between the verbs *"commit"*, *"modify"*, and *"change"* (*e.g., "What developers change...?"*). We explained to them after the study that the three verbs are treated as synonyms—they are compiled into exactly the same SPARQL query—and that the purpose of the synonyms is to provide multiple ways of formulating a query, to achieve a similar level of freedom as with full natural language input. Their response was that they would have appreciated a short demo of HAWKSHAW. Nonetheless, both of the subjects performed reasonably well despite their limited knowledge of the details of our approach.

Task 5: Finding exceptional entities in terms of changes—The file that has changed most often in the past?

Results. Both groups performed equally poor in solving Task 5. None of the control group and only three out of 18 subjects of the experimental group obtained the correct solution. The small difference between the groups lead the MWU test to retain the null-hypothesis concerning correctness. For completion time, the test yielded a significant difference in favor of the experimental group. However, this was simply a consequence of the experimental group's premature assumption that they had provided the right answers. The difficulty of the task was perceived high by the experimental group and very high by the control group. The first group assessed the degree of realism as high, the second one as rather low. Both differences in ratings were statistically significant.

Interpretation. Succeeding with only the baseline tools required some familiarity with SVN's numbering scheme for revisions: with the Eclipse Subversive plug-in, the version numbers are displayed next to each directory or file name. Searching first for the directory with the highest number and then checking its files' version numbers would have quickly yielded the correct answer. To obtain the correct answer with HAWKSHAW, one had to enter, *e.g.,* the question *"What file has the most versions?"* Interestingly, nine out of 18 subjects of the experimental group gave the right answer to the wrong question, *i.e.,* they wrote down the file that was changed by the most developers instead. The reason for this misunderstanding was that they relied too much on the guidance provided by our approach. When they wanted to enter *"What file changed...?"*, HAWKSHAW instead proposed *"What file is changed by..."*. The only way to complete the latter sentence was by choosing *"...the most developers?"*. Instead of questioning the result and then reformulating their information need, they accepted the non-applicable answer and proceeded to the next task. We attribute this shortfall to the study setting and claim that, in a real setting, a developer would not be satisfied with the answer and further strive for an appropriate answer.

Task 6: Learning about the changes of other team members—The last five files changed by *Developer*$_1$?

Results. HAWKSHAW performed as well as the baseline tools in Task 6, with no significant differences according to the results of the MWU test. The mean correctness of the control group was observed slightly higher than that of the experimental group. We therefore, again, binned the answers into correct and incorrect ones, and ran the Chi^2 test on the data. The test this time clearly retained the null-hypothesis at a

confidence level of 99% with a p-value of 0.229. The difficulty was perceived rather high in both groups, whereas the realism was rated rather high, with an insignificantly lower rating among the participants of the control group.

Interpretation. We were surprised to see that the control group could not outperform the experimental group significantly in this task. Our expectations were grounded on the development history of Apache Ivy and the concrete developer we named in the task description. When one opened the Eclipse *History View* on the project, the author of the five most recent commits (which were therefore displayed on top) was the one we were looking for and the corresponding files could be found by clicking through the first few table items. In contrast to the baseline tools, solving this question with HAWKSHAW was particularly difficult because the subjects first had to obtain all the files changed by the particular developer. They then had to sort the files by their modification dates in the result view. Multiple participants therefore later said that they did not understand how to sort the results properly and would have wished for a thorough example in the tutorial related to this functionality.

Task 7: Becoming familiar with a team member's work—What feature requests were implemented by *Developer*$_2$?

Results. No statistical differences in correctness and completion time between both groups could be observed for this task. Most subjects found the task realistic but rather difficult to solve.

Interpretation. Similar to the last task, this task was concerned with team awareness—but this time from an issue-perspective. Mapping the task description to a search strategy asked for some creativity because there is no direct way to obtain the feature request implemented by a given developer. However, finding any closed feature requests with the appropriate assignee provides a good approximation, which can then be verified against the change log. The control group had to use a tool external to the IDE to solve the task: the Web front-end of JIRA. The simple search of JIRA provides an input mask to query explicitly for feature requests with a given assignee and resolution, rendering the task straight-forward. The experimental group had to reformulate the task to a query containing an adjective, *i.e., "What **closed** feature requests are assigned to...?"*. We did not mention the possibility of using adjectives in the HAWKSHAW tutorial, yet most subjects of the experimental group were able to obtain the correct answer.

Task 8: Finding issue reports relevant to other team members—The issues *Developer*$_3$ and *Developer*$_4$ commented on?

Results. None of the subjects of the control group were able to solve the task correctly, whereas those of the experimental group succeeded without exception in under two minutes. As a consequence, the control group found the task very difficult and rather unrealistic—in contrast to the experimental group, which rated the same task significantly different: very easy and rather realistic.

Interpretation. The task was neither solvable with the JIRA Web front-end, nor from within the IDE. However, the control group could have run a straight-forward Google query to obtain the correct answer within seconds. It seems that they did not think of this possibility and tried instead to submit queries with the advanced search of JIRA until the time for the task ran off. The HAWKSHAW group, on the other hand, did not report any problems.

The ratings on the relevance of the tasks show some disagreement. It has to be mentioned however, that the task was derived from the literature [Fritz and Murphy 2010] and that there exist JIRA plug-ins, as well as a feature request for JIRA itself with more than 50 votes to support this particular information need. We therefore consider the task relevant, although the control group disagreed with us.

Task 9: Finding issues affected by a particular blocker—The issues blocked by $Issue_1$?

Results. While correctness was not an issue for either group, the subjects with HAWKSHAW were significantly faster than those using the JIRA Web front-end. Both groups found the task very realistic, but there was a statistically significant difference in the perceived difficulty.

Interpretation. The control group was about three times slower than the experimental group. The difference is worth emphasizing because the Web front-end of JIRA contains a clearly visible section *Issue Links* near the top of the overview page of each issue. There, the issues blocked by the current one (including the word *"blocked"*) are listed. On the other hand, the excellent performance of the experimental group shows how well the subjects adapted to our quasi-natural language interface after only a few tasks.

Task 10: Finding the Java classes affected by a particular issue—The classes affected by $Issue_2$?

Results. For the tenth task, we again observed a significant difference in completion time between the two groups. The difference in correctness was insignificant for the MWU test. The Chi^2 test retained the null-hypothesis that the number of fully correct answers depends on the group at a confidence level of 99% but rejected it at 95% with a p-value of 0.012. The difference observed for the ratings concerning realism was insignificant with MWU at a confidence level of 99%. However, it is significant at 95% (p-value=0.027). For the difficulty ratings, we observed a significant difference, even without adjusting the α-level.

Interpretation. JIRA integrates with SVN to show the files that were committed to resolve an issue. This functionality was illustrated in the tutorial given to the control group. The file diffs could then be viewed online by the subjects to verify whether the changes really applied to the top-level class declared within that file. Despite this well-integrated approach, HAWKSHAW outperformed the baseline tools by far with regard to completion time. The statistical results on correctness are inconclusive, but slightly in favor of HAWKSHAW when we incorporate Chi^2 with a confidence level of 95%.

Task 11: Finding all the issues affecting a particular Java class—The issues that affected $Class_4$?

Results. The task was solved faster and with a higher correctness by the experimental group than by the control group. This advantage is statistically significant. The control group found the task rather realistic on average, but rather difficult to solve. The experimental group was convinced that the task is very realistic and that it is very easy to solve with our approach. The difficulty ratings are significantly different. For realism, the differences are significant at a confidence level of 95%, but not at a level of 99% (p-value=0.03).

Interpretation. The subjects of the control group had to manually search through the SVN log of the class for issue numbers in the commit messages. Their difficulty rating indicates that this is a tedious task—and also error-prone, as it is easy to miss a reference to an issue. We generally noticed that the subjects often provided incomplete solutions with the baseline tools, whereas those in the experimental group in most cases provided either no solution or a fully correct one. This is reflected in the many high median correctness values of 1.0 listed in Table IV. The effect is also apparent in Figure 10, where the stacked bar charts show for both groups and for each task the correct and partially correct answers in percent of the total number of answers we received. The left chart shows the results for the experimental group; only in four out of 13 tasks, partially correct answers were given, as opposed to the control group, shown in the right chart, where partially correct answers were given in nine out of 13 tasks. The observed phenomenon is grounded in the nature of our approach: if users are able to correctly formulate their question, then recall will most likely be at 100%.

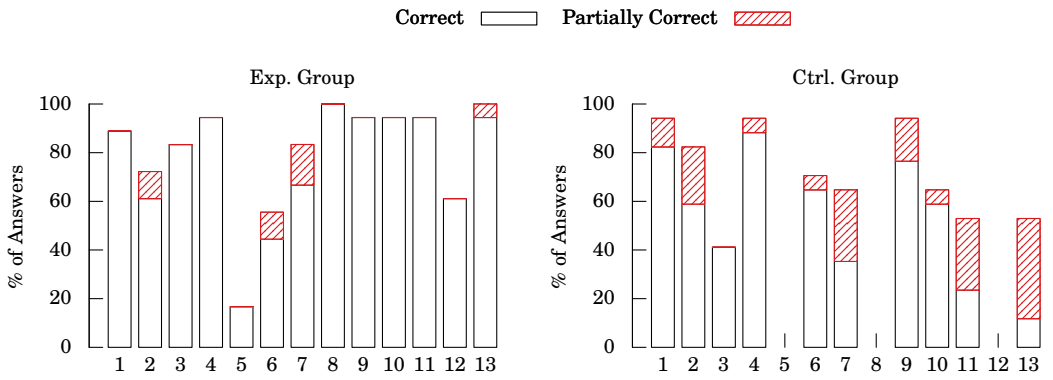

Fig. 10. Percentage of correct and partially correct Answers per Group and Task

Task 12: Finding exceptional entities in terms of error-proneness—The most error-prone class?

Results. No participant in the control group was able to solve Task 12 correctly in time, which corresponds to the difficulty ratings. The experimental group performed reasonably well in terms of correctness and completion time, but still perceived the task as difficult. The groups, however, both agreed that the task was realistic: the experimental group found it very realistic, the control group rather realistic. The difference between their difficulty ratings is significant; the control group found the task more difficult to solve than the experimental group.

Interpretation. One difficulty that participants from both groups reported was that they did not understand the term *"error-prone"* as non-native English speakers. A few users of HAWKSHAW were thus not able to translate the task description into a meaningful question with a subject-predicate-object structure. The baseline tools did not provide any means to quickly solve the task, but rather it was necessary to skim the SVN log of the whole project for issue numbers and associated files. This was rather unrealistic in the short amount of time given to the subjects. Despite the unfavorable situation for the baseline tools, we can still conclude that the users of HAWKSHAW were able to solve a task reasonably well that was deemed important by the majority of the participants of our study, as well as by other researchers [Kim et al. 2007].

Task 13: Finding issues that affect multiple Java classes—The issues that affected $Class_5$ and $Class_6$?

Results. Except for one outlier, the maximum correctness was achieved by all subjects of the experimental group. Nine out of 17 subjects of the control group either provided a correct answer or at least a partially correct one. On average, the control group nearly maxed out the available time, whereas most members of the experimental group were able to solve the task in less than one and a half minutes. Statistically significant differences between the groups were also observed for the difficulty assessment: the experimental group found the task easy to solve, the control group rather difficult. Both groups agreed that the task was rather realistic. Because Task 13 was very similar to Task 11, we tested whether there was a difference in the performance per group between the tasks. For the experimental group we determined with the Related-Samples Wilcoxon Signed-Rank test that there was no significant increase in completion time (p-value=0.396, α=0.01) between Task 11 and 13. For the control group, the increase was significant—at least at the confidence level of 95%, with a p-value of 0.038. We observed no significant difference in correctness for either group.

Interpretation. To conclude our study, we re-iterated on a previous task, Task 11, but asked for the intersection among the answers to two distinct questions this time. The SVN logs that the subjects of the control group had to go through were comparable in size for both tasks, but for Task 13, the number of relevant commit messages doubled because two, instead of one, Java classes were involved. As we can see from the results, the influence of the additional twist on the correctness was negligible for both groups but there was a negative impact on the completion time for the control group only. Because the composition of different information fragments happens implicitly and therefore automatically in HAWKSHAW, users with HAWKSHAW were much faster than those with the baseline tools.

4.10. Summary of Results

The overall results presented in Section 4.8 showed that with HAWKSHAW, users were able to solve the body of tasks in our study with a significantly higher correctness in a significantly shorter amount of time. In Section 4.9 we have analyzed which tasks contributed the most to the superiority of our approach.

The analysis of the individual tasks showed that, for the Tasks 5, 8, and 12, none of the subjects were able to provide a fully correct solution with the baseline tools. As a consequence, they used up the total amount of time allotted to that task. To test whether only these three tasks were responsible for the significant difference in correctness and completion time, we excluded them from our data and re-ran the MWU test for the adjusted overall results at a confidence level of 99% with an α of 0.01. The null-hypothesis concerning the correctness was nonetheless rejected with a p-value of 0.0048; similarly the null-hypothesis for completion time, but with a p-value of 1.47E-5. In Tasks 1 to 7, both groups achieved a similar correctness. TIt was possible for the control group to obtain good scores in terms of correctness in the individual tasks, which suggests that our selection of tasks in general did not penalize the baseline tools unduly.

For five tasks (Tasks 3, 9, 10, 11, and 13) we could observe a significant reduction of completion time when HAWKSHAW was used, rather than the baseline tools. In some cases, the average time savings were as high as 300%. Furthermore, the experimental group was able to solve Task 8 and 12 within the time limit of five minutes, whereas the control group was not. Unsurprisingly, the largest time savings comes from the cross-domain tasks, but HAWKSHAW also yields a significant benefit where the composition of data from different tools or multiple queries is crucial for solving the task at hand quickly. We emphasize that our approach performed as good as the mature code search features of Eclipse.

Besides these gains in efficiency, another main advantage of our approach lies in the significant usability improvements reported by the subjects. The individual responses to the post-task usability questionnaire SEQ support the excellent SUS score achieved by HAWKSHAW: The subjects of the experimental group perceived eight tasks significantly easier than the members of the control group (*i.e.,* Tasks 3, 5, 8, 9, 10, 11, 12, 13). The remaining five tasks were rated similarly by both groups—including the ones related to source code search. That our approach stands up against Eclipse in the source-code-centric tasks not only in terms of completion time but also in terms of usability is especially remarkable because it were possible to solve the tasks in Eclipse with only two mouse-clicks or by pressing a single shortcut.

We are convinced that we have not yet fully exploited the potential of the HAWKSHAW approach: with more sophisticated grammar rules and additional synonyms encoded into our ontologies, there is even more room for improvement.

4.11. Threats to Validity

Internal Validity is concerned with uncontrolled factors that may have biased our results in favor of either the experimental or the control group. One concern is that the subject's experience level might have been distributed unevenly over the two study groups. To mitigate this threat, we have randomly assigned the participants to the experimental and control group. Figure 5 illustrates further that this concern is unjustified, *i.e.,* that the distribution of experience and skills was fair.

For measuring completion time, we provided the participants with an experiment timer application as part of the Eclipse IDE. Since each participant himself was responsible for reporting time correctly (*i.e.,* starting and stopping the timer, as well as writing down the number of elapsed minutes and seconds after each task), it is possible that incorrect or even completely fictitious task times have been reported by some of the subjects. However, we carried out inspections at random and, in principle, the design of our study did not motivate the participants to game the system.

Lott and Rombach [1996] mention that having a subject perform several related tasks within a short time interval may cause non-negligible learning effects. Learning effects may have affected the results on correctness and completion time because the experimental group could rely solely on HAWKSHAW for each of the task, whereas he subjects of the control group had multiple tools at their disposal. Depending on the task, they had to use of a different tool to obtain the correct answers and could therefore benefit less from potential learning effects. While the exact strength of the potential learning effect remains unknown, we argue that the related threat to validity is relatively low for Tasks 1 to 6 and 10 to 13. For the first six tasks, the participants of the control group benefited from several months or even years of experience in using the Eclipse IDE, including the Java search features and the history view. The users of HAWKSHAW, however, had never used the quasi-natural language approach before. In Tasks 10 to 13, the subjects of the control group could reuse the same tools again and in a similar fashion as in the previous tasks. The cross-domain nature of these tasks simply required an additional step to compose the partial results obtained from each tool. Learning effects might have had a stronger impact on the results for Tasks 7 to 9, which were related to bug and issue management. The participants knew the concept of issue trackers in principle, but barely any of them had worked with the JIRA issue tracker before, and the tutorial we provided them could only account for this lack of experience to a certain extent.

Regarding *construct validity*, it is possible that the questionnaires used to measure usability are not an adequate means for assessing usability of the baseline tools or our approach. Given that the SUS and SEQ were used successfully in thousands of different usability assessments in various application domains and that numerous researchers attribute an excellent reliability to it, we believe that this threat is relatively low. Furthermore, we calculated a very high Cronbach alpha value for the SUS scores observed in our study, indicating that the SUS measured consistently. It has also been shown that the SUS does not provide a strong correlation to task-level metrics. Task completion rates, for example, explain only around 5% to 6% of the changes in SUS scores [Sauro and Lewis 2009]. This leaves us confident that we were able to measure overall system satisfaction mostly independent from the choice of our tasks.

Threats to *conclusion validity* may arise from too few participants in our study. To alleviate this threat—and since we could not safely assume a normal distribution for our data—we selected the non-parametric MWU test which also works on non-normally distributed data and is robust even against small sample sizes. We complemented the MWU test with a thorough visual analysis in box plots, histograms, Q-Q plots, and where appropriate, with other statistical test procedures.

External Validity is the degree to which the results can be generalized and transferred to other situations. In that respect, our subjects may not have been a representative sample for junior developers employed in industry. However, we only invited students that were studying in the fourth semester of the Bachelor's level or higher. Thus they had successfully passed multiple courses on programming, algorithms and data structures, as well as on software engineering methodologies. Many of them had already completed an internship in a software development company and only a few more courses plus their Bachelor's thesis were left before they would begin their careers in industry.

None of the participants were familiar with Apache Ivy—the software system we used in our study. Developers with more knowledge of the system might work differently or be able to apply more effective strategies to use their existing tools. The situation that participants faced in our study, however, is comparable to that of developers newly joining a software project.

We only performed our study with one object-oriented system, which may not be representative for the family of object-oriented systems in industry. Especially for the cross-domain tasks, we strongly relied on the rigid change process of the Apache Foundation, in order to be able to reliably link issues to changes. In projects for which no such software evolution data is available, the completeness of the results returned by HAWKSHAW will suffer and consequently, its users will perform less effectively. However, in such a scenario, also the users of the baseline tools will be challenged. Another selection of baseline tools may provide more competitive features and a higher usability than the one we have chosen.

The tasks chosen for our study might not be realistic or they might favor HAWKSHAW and therefore the results might not generalize to others than the tools used in the study. We re-used existing catalogues of common developer questions that were compiled by other researchers through surveys and interviews of practitioners, but we had to adapt most questions to make them more concrete. We also extended some of the original questions to assess certain facets of our approach, and thus deviated further from the original questions. To control the effects of our modifications, we asked the subjects to assess the practical relevance of our tasks. Most tasks were clearly considered by both groups as being realistic. For the few instances where the subjects were at odds with each other, we provided a thorough discussion on why we still are convinced of the relevance of our selection. However, the subjects' ratings concerning practical relevance still need to be treated with caution: we detected a modest correlation ($r = 0.4$) between difficulty and practical relevance, *i.e.,* those subjects that rated a task more difficult also tended to attribute a lower practical relevance to it. The exact causal chain remains unknown and further investigations are necessary, but it is possible that the practical relevance ratings were confounded by difficulty and therefore cannot support the generalizability of our approach.

The baseline we compared HAWKSHAW against was comprised of a common IDE with SVN support, the Web front-end of a widely used issue tracking tool, as well an internet browser to access the issue tracker and to perform Web searches. Our selection of tools might not have taken into account that development teams in industry might use even more powerful programs.

4.12. Discussion and Synthesis

The user study focussed on less experienced subjects since we expected our approach to be especially apt for novice developers, not yet deeply familiar with the tools available. The reasoning behind this was that novice developers will most likely not know about most of the advanced features of an IDE. A single point of access for their information needs would relieve them from browsing all menus and dialogs to find what they are

looking for, from reading help pages, and so on. The positive feedback received from some of the more seasoned participants provides anecdotal evidence that the results of the study can be generalized to a broader target group of users. Further investigations are needed to support this claim with scientific evidence.

The HAWKSHAW framework revolves around a knowledge base built with Semantic Web technology and a quasi-natural language interface. The Semantic Web yet struggles to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (*e.g.,* in [Kupershmidt et al. 2010]). The same accounts for natural language interfaces, which have been mostly neglected in software evolution research so far, but recently gained momentum in other domains. Popular examples are Apple's Speech Interpretation and Recognition Interface (Siri),[8] the Wolfram Alpha answer engine developed by Wolfram Research,[9] and IBM's Watson computer system for answering natural language questions [Ferrucci 2011].

The overall conclusions we can draw from the strong empirical results found in our user study is that both the Semantic Web and natural-language interfaces exhibit significant potential for building the next generation of software engineering support tools. Such technology should therefore be at least considered whenever researchers in the field of software engineering devise approaches that involve knowledge representation and developer-computer interfaces. A quasi-natural language interface such as the one incorporated in the HAWKSHAW framework requires basically no learning effort from its users and therefore can accelerate the adoption of novel research tools in practice.

The design of our study and particularly the task selection take into account the two aspects mentioned above: we evaluated the quasi-natural language interface of HAWKSHAW against different classical user interfaces, but we also also looked at the advantage of the integrated view of our knowledge base over an heterogenous tool landscape where users are exposed to multiple sources of information.

The first three tasks of our study were selected to compare HAWKSHAW with the Java search of Eclipse, which is based on a classical, context-menu-driven user interface. Remarkably, users were as effective with the natural language interface as the with the code search, although the latter is much more specific and therefore highly optimized to these kind of tasks. While there is no indication that in replacing the current code search with HAWKSHAW we would gain benefits in terms of effectiveness for simple code search tasks, HAWKSHAW can still complement the existing features by providing an entry point to developers not yet familiar with the IDE. Furthermore, a query-language-based approach such as ours provides additional expressiveness when it comes to solving more complex tasks. Thus it can relieve users from manual composition of the results obtained from multiple invocations of a menu-driven search. This claim is supported by the results for Task 3 on time efficiency.

A similar observation can be made for the tasks related to the development history where users of the baseline tools had to browse a table-based view. Such views sometimes implement simple keyword-based search and basic filtering, but these search widgets usually lack of means to express relationships between data. In our study, users of the baseline tools consequently spent much more time for going through each of the entries of the table view than the users of HAWKSHAW, who generally succeeded in clearly specifying their information needs with a concise query.

Subjects of the control group had to use the Web-front-end of the issue tracker for solving the Tasks 7 to 9. The front-end provided Web forms for querying and displayed the results as hypertext documents. Many subjects of the control group overlooked

[8]http://www.apple.com/iphone/features/siri.html
[9]http://www.wolframalpha.com/

very prominent information, such as the blockers of certain issues, which suggests that they were overwhelmed by the sheer amount of information displayed. This may be attributed to their unfamiliarity with JIRA and users are likely to overcome these issues after a training phase, but yet the advantage of HAWKSHAW lies in its high learnability—users do not need to familiarize themselves with another user interface paradigm when working with bug reports, other than the one they already use for searching in code, querying the development history, and so on. Notable are also the problems that subjects faced when answering Task 8 where they had to find the issues on which two particular developers had commented in the past. JIRA did not explicitly support searching for issues a person has commented on, so the participants had to fall back to a regular Web search for obtaining the correct answer. With the HAWKSHAW framework, there is no need to implement such specific searches. Instead, data is simply described through an OWL ontology, along with the general grammatical structure of the queries, and consequently even queries will work that were not explicitly foreseen. Since describing data with OWL is not much different from defining a relational database schema, the overhead of doing so is negligible compared to classical approaches.

Whereas the previous tasks focussed on comparing different user interface paradigms against HAWKSHAW, the last four tasks put emphasis on the advantage that HAWKSHAW's integrated knowledge base yields over multiple, poorly integrated sources of information. The clear results for these tasks indicate that the manual composition of different information fragments is laborious and error-prone, and that our approach can provide significant relief in this regard. In synergy with the quasi-natural language interface, the knowledge base becomes a powerful tool even for novice users. That means they get a query language whose expressivity is comparable to formal query languages but overcomes the initial hurdle of learning a specific syntax and vocabulary.

The HAWKSHAW approach is not without its limitations. We demonstrated through our study that a surprisingly small set of static grammar rules and synonyms allows for a variety of different queries. However, additional investigations are needed to identify variations in the exact phrasing of conceptual queries that might occur when software engineers formulate their information needs in practice. The findings then need to be encoded in terms of static grammar rules. Furthermore, the natural language annotations (synonyms) of SEON are based on our personal vocabulary. This vocabulary might be biased towards the programming languages and tools we regularly use and therefore fail to adequately describe the concepts which developers with a different background are familiar with. In this context, we evaluated the use of general-purpose lexical databases of English, in particular the WordNet database [Miller 1995], to increase the vocabulary of HAWKSHAW with additional synonyms. However, for our approach, such databases have proven themselves unsuitable. The problem we encountered was that many technical terms also have non-technical meanings in daily life. For example, the term "Method" from in object-orientated programming has synonyms such as "adjustment," "approach," "fashion," etc. If we automatically add those to the list of proposals presented by our query interface, then the developers are no longer restricted to reasonable questions, *i.e.,* they can then enter completely meaningless ones such as *"What fashion invokes the approach foo()?"* Sridhara et al. [2008] have reported similar issues when they applied linguistic tools to source code and other software artifacts. The authors propose to augment WordNet with relations specific to software, which could also be valuable for improving the HAWKSHAW approach further.

Currently, HAWKSHAW is not particularly well-suited for answering questions related to time intervals or specific points in time. For example, questions such as *"What classes were changed yesterday?"* or *"What bugs were fixed between May and August?"* cannot be entered directly. However, it is possible to query, *e.g.,* for all bug fixes and then sort the results by their fix date. This puts HAWKSHAW on a par with many tools used in

practice, but formal query languages would clearly have an edge over our approach in that respect (at the cost of additional learning effort).

While it is notable that existing catalogues of common developer questions rarely contain examples such as the ones mentioned above, we still know from our own experience that they occur frequently in practice, so that an adequate support is desirable. While the static grammar of HAWKSHAW can be extended to incorporate corresponding natural language rules, more research is needed to come up with an appropriate translation into SPARQL. Approaches such as Temporal Reasoning could provide a solution to this issue [Tappolet 2011].

Currently, textual or keyword-based search is unsupported because of the guided nature of the query composition approach. In consequence, users cannot search, for example, for all the files that contain the word "database." This is sufficiently well supported by existing tools so that we, in principle, see no immediate need for action. However, to preserve the idea of a single point of access for common information needs, it would still make sense to add special non-terminal symbols to the static grammar rules that would temporarily switch the query interface from a guided mode into one that allows for entering free-form text (*e.g.,* when entering opening quotation marks until closing ones are typed). The translation into SPARQL is then straight-forward thanks to built-in regular expression support of the language.

5. RELATED WORK

LaSSIE was an early attempt to integrate multiple views on a software system in a knowledge base [Devanbu et al. 1991]. It also provided semantic retrieval through a natural language interface. Frame systems, a conceptual predecessor to the ontologies of the Semantic Web, were used to encode the knowledge. The aim of LaSSIE was to preserve knowledge of the application domain for maintainers of the software system. HAWKSHAW does not yet incorporate any application-specific knowledge but focuses on answering common developer questions related to software evolution.

Hill et al. [2009] presented an algorithm to extract noun, verb, and prepositional phrases from method and field signatures in source code to enable *contextual searching*. The queries they support are closer to keyword search on identifiers found in source code than to full natural language questions and they do not cover structural information, such as caller-callee or inheritance relationships among source code entities. In contrast to ours, the approach completely neglects history or bug and issue related queries.

Another promising approach for querying source code with natural language queries was introduced by Kimmig et al. [2011]. Their query interface uses part-of-speech tagging and stemming to enable free-form queries, while our approach guides developers during query composition and does not rely on any natural language processing. In consequence, it is possible to enter queries, which are not understood by the approach of Kimmig *et al.*—unlike HAWKSHAW, which prevents its users from composing unrecognizable queries. Furthermore, we support querying of multiple facets of software evolution, whereas Kimmig *et al.* did not report whether their approach can be generalized to domains other than that of static source code.

Many approaches have been proposed that use specific languages to query software artifacts. They are either based on standard database languages, such as SQL or Datalog (*e.g.,* CodeQuest [Hajiyev et al. 2006]), customized Prolog implementations (*e.g.,* JQuery [Janzen and Volder 2003] or ASTLog [Crew 1997]), or a custom language (*e.g.,* SCA [Paul and Prakash 1996]). Their aim is to help developers in effectively exploring and better understanding code, uncovering information that would be impossible or extremely hard to find with standard tools. However, most of them require the user to master syntax and vocabulary of a specific query language. Our approach guides developers in vocabulary, as well as in syntax, to construct well-formed and coherent

questions about different aspects of a software system. To the best of our knowledge, most of these works are limited to source code queries and no other approach exists that supports multiple software evolution facets.

The *Sphere Model* by de Alwis and Murphy [2008] enables composition of different sources of information to implement conceptual queries. In our previous work, we have shown that HAWKSHAW is able to answer their queries that are related to static source code information and, in addition, our approach allows developers to formulate their questions with quasi-natural language. Thanks to the recent improvements of our approach, we can now also answer a broader range of evolution-related queries.

Fritz and Murphy [2010] presented the *Information Fragment Model* that, similar to the previously mentioned *Sphere Model*, supports the composition of information from multiple sources, as well as the presentation of the composed information. However, the model does not support explicit querying, but rather allows for the ad-hoc combination of results from two queries obtained with other approaches, based on identifier or text matching.

The work by Kaufmann and Bernstein [2010] is unrelated to the field of software engineering research. However, they presented a usability study of query interfaces with 48 users. The study incorporated geographical data encoded in an OWL knowledge base and four query interfaces featuring four different query approaches. The goal was to demonstrate the usefulness of natural language interfaces for casual end-users. One of the evaluated interfaces was Ginseng, which our approach was originally based on. The conclusion drawn from the experiment was that, with natural-language questions, *"users can communicate their information need in a familiar and natural way without having to think of appropriate keywords in order to find what they are looking for."* The authors also found empirical evidence that *"people can express more semantics when they use full sentences and not just keywords."* The results from the HAWKSHAW user study suggest that these insights are generalizable from geographical data and casual end-users to both software developers and the domain of software evolution and maintenance.

6. CONCLUSIONS

Nowadays, the sheer scale of many industrial software development projects demands a wide range of tools to support processes and enable collaboration. Mastering these tools to such an extent so that one can answer common information needs that arise during daily development tasks is challenging and puts a high cognitive load on developers.

In this paper, we have shown that quasi-natural language interfaces provide a valuable alternative to menu-driven search with modern IDEs and project tracking tools. We argued that our HAWKSHAW approach is helpful in solving various tasks related to software evolution and maintenance, and that it also scales to real industrial-size software systems. To support our claim, a user study with 35 subjects was conducted. In summary, the results of our study provide empirical evidence that:

— Overall, developers achieved a significantly higher correctness when solving common software evolution tasks with our quasi-natural language approach than with a baseline of more traditional tools, such as a common Java IDE and the Web front-end of a popular bug and issue tracker. Looking further at the results for each task individually, we observed that users of HAWKSHAW always achieved at least the same level of correctness as their counterparts with the baseline.
— Our approach leads to a significant improvement in time efficiency. Overall, we have seen time savings of 35.41% when compared to the baseline, with gains of up to 300% in some individual cases.

— The overall system satisfaction of users with HAWKSHAW is clearly better than that of the baseline's users. The subjects rated our approach on average with a total score of 74.31 on the System Usability Scale, whereas the baseline only achieved an average rating of 38.24. The high score of HAWKSHAW is directly related to its high usability and learnability.

Our approach serves as a single point of access to facts about source code and various other knowledge which is otherwise hardly integrated and locked away in project trackers or version control systems. Because it is based on quasi-natural language, getting familiar with the HAWKSHAW interface requires little to no learning effort. It can be easily extended with additional grammar rules, synonyms then available to developers during query composition, and even with whole new software engineering domains—solely by using standardized means of knowledge engineering.

Future work will focus on the extension of HAWKSHAW's querying capabilities through the means described above, as well as on conducting a field study with professional developers in industry. From that, we hope to gain deeper insights on the potential and limitations of our query framework in practice.

ACKNOWLEDGMENTS

The work presented in this paper was supported by the Swiss National Science Foundation as part of the "Systems of Systems Analysis" (200020_132175) project. The authors are grateful to Serge Demeyer, Matthias Hert, and in particular the anonymous reviewers for their valuable comments that greatly helped to improve the paper.

REFERENCES

BANGOR, A., KORTUM, P. T., AND MILLER, J. T. 2008. An empirical evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction 24,* 6, 574–594.

BASIT, H. A., ALI, U., AND JARZABEK, S. 2011. Viewing simple clones from structural clones' perspective. In *Proceedings of the 5th International Workshop on Software Clones*. 1–6.

BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. 1998. RFC 2396 - uniform resource identifiers (URI). IETF RFC. `http://www.ietf.org/rfc/rfc2396.txt`.

BERNERS-LEE, T., HENDLER, J., AND LASSILA, O. 2001. The Semantic Web. *Scientific America 284,* 5, 34–43.

BERNSTEIN, A., KAUFMANN, E., KAISER, C., AND KIEFER, C. 2006. Ginseng: A guided input natural language search engine for querying ontologies. In *Jena User Conference*. 2–4.

BROOKE, J. 1996. SUS - a quick and dirty usability scale. In *Usability Evaluation in Industry*, P. W. Jordan, Ed. Taylor & Francis, 189–194.

BROOKS, JR., F. P. 1995. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

CHOWDHURY, G. G. 2004. *Introduction to Modern Information Retrieval* 2nd Ed. Facet, London, UK.

CREW, R. F. 1997. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*. 229–242.

DE ALWIS, B. AND MURPHY, G. C. 2008. Answering conceptual queries with ferret. In *Proceedings of the 30th International Conference on Software Engineering*. 21–30.

DEAN, M. AND EDS., G. S. 2004. *OWL Web Ontology Language Reference*. W3C Recommendation. `http://www.w3.org/TR/owl-ref/`.

DEVANBU, P., BRACHMAN, R., AND SELFRIDGE, P. G. 1991. LaSSIE: a knowledge-based software information system. *Communications of the ACM 34,* 5, 34–49.

FERRUCCI, D. A. 2011. IBM's Watson/DeepQA. In *Proceedings of the 38th annual international symposium on Computer architecture*.

FLURI, B., WÜRSCH, M., PINZGER, M., AND GALL, H. C. 2007. Change Distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering 33,* 11, 725–743.

FRITZ, T. AND MURPHY, G. C. 2010. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. 175–184.

GRUBER, T. R. 1993. A translation approach to portable ontology specifications. *Knowledge Acquisition 5,* 2, 199–220.

HAJIYEV, E., VERBAERE, M., AND DE MOOR, O. 2006. CodeQuest: Scalable source code queries with datalog. In *ECOOP 2006 – Object-Oriented Programming*. Lecture Notes in Computer Science Series, vol. 4067. Springer Berlin / Heidelberg, 2–27.

HALLETT, C., SCOTT, D., AND POWER, R. 2007. Composing questions through conceptual authoring. *Computational Linguistics 33,* 1, 105–133.

HATTORI, L., D'AMBROS, M., LANZA, M., AND LUNGU, M. 2011. Software evolution comprehension: Replay to the rescue. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*. 161–170.

HENNINGER, S. 1994. Using iterative refinement to find reusable software. *IEEE Software 11,* 5, 48–59.

HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st IEEE International Conference on Software Engineering*. 232–242.

HOLM, S. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics 6,* 2, 65–70.

JANZEN, D. AND VOLDER, K. D. 2003. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. 178–187.

KAUFMANN, E. AND BERNSTEIN, A. 2010. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web 8,* 4, 377–393.

KIM, S., ZIMMERMANN, T., WHITEHEAD JR., E. J., AND ZELLER, A. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*. 489–498.

KIMMIG, M., MONPERRUS, M., AND MEZINI, M. 2011. Querying source code with natural language. In *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering*. 376–379.

KLYNE, G. AND EDS., J. J. C. 2004. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

KO, A. J., DELINE, R., AND VENOLIA, G. 2007. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*. 344–353.

KUPERSHMIDT, I., SU, Q. J., GREWAL, A., SUNDARESH, S., HALPERIN, I., FLYNN, J., SHEKAR, M., WANG, H., PARK, J., CUI, W., WALL, G. D., WISOTZKEY, R., ALAG, S., AKHTARI, S., AND RONAGHI, M. 2010. Ontology-based meta-analysis of global collections of high-throughput public data. *PLoS ONE 5,* 9.

LATOZA, T. D., VENOLIA, G., AND DELINE, R. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*. 492–501.

LEWIS, J. R. 1992. Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ. In *Proceedings of the Human Factors Society*. 1259–1263.

LEWIS, J. R. AND SAURO, J. 2009. The factor structure of the system usability scale. In *Proceedings of the 1st International Conference on Human Centered Design*. 94–103.

LOTT, C. M. AND ROMBACH, H. D. 1996. Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering 1,* 3, 241–277.

MILLER, G. A. 1995. WordNet: a lexical database for english. *Communications of the ACM 38*, 39–41.

NORUSIS, M. 2010. *SPSS 18 Advanced Statistical Procedures Companion*. Prentice Hall, New Jersey, NY, USA.

PAUL, S. AND PRAKASH, A. 1996. A query algebra for program databases. *IEEE Transactions on Software Engineering 22,* 3, 202–217.

POSNETT, D., FILKOV, V., AND DEVANBU, P. 2011. Ecological inference in empirical software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. 362–371.

POWER, R., SCOTT, D., AND EVANS, R. 1998. What you see is what you meant: Direct knowledge editing with natural language feedback. In *Proceedings of the 13th Biennial European Conference on Artificial Intelligence*. 675–681.

PRUD'HOMMEAUX, E. AND EDS., A. S. 2008. SPARQL query language for RDF. W3C Recommendation. `http://www.w3.org/TR/rdf-sparql-query/`.

SAURO, J. 2010. If you could only ask one question, use this one. Personal Blog. `http://www.measuringusability.com/blog/single-question.php`.

SAURO, J. 2011. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. CreateSpace Independent Publishing Platform.

SAURO, J. AND DUMAS, J. S. 2009. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*. 1599–1608.

SAURO, J. AND LEWIS, J. R. 2009. Correlations among prototypical usability metrics: evidence for the construct of usability. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*. 1609–1618.

SILLITO, J., MURPHY, G. C., AND DE VOLDER, K. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 23–34.

SILLITO, J., MURPHY, G. C., AND VOLDER, K. D. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering 34,* 4, 434–451.

SIRIN, E., PARSIA, B., GRAU, B. C., KALYANPUR, A., AND KATZ, Y. 2007. Pellet: A practical OWL-DL reasoner. *Journal Web Semantics 5,* 2, 51–53.

SRIDHARA, G., HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2008. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*. 123–132.

TAPPOLET, J. 2008. Semantics-aware software project repositories. In *Proceedings of the European Semantic Web Conference, Ph.D. Symposium*.

TAPPOLET, J. 2011. Managing temporal graph data while preserving semantics. Ph.D. thesis, University of Zurich.

THOMPSON, C. W., PAZANDAK, P., AND TENNANT, H. R. 2005. Talk to your semantic web. *IEEE Internet Computing 9,* 6, 75–78.

WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B., AND WESSLÉN, A. 2000. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA.

WÜRSCH, M., GHEZZI, G., HERT, M., REIF, G., AND GALL, H. C. 2012. Seon - a pyramid of ontologies for software evolution and its applications. *Computing 94,* 11, 857–885.

WÜRSCH, M., GHEZZI, G., REIF, G., AND GALL, H. C. 2010. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. 165–174.